Play framework documentation

Welcome to the play framework documentation. This documentation is intended for the **1.1 release** and may significantly differs from previous versions of the framework. Check the version 1.1 release notes.

> The 1.1 branch is in active development.

Getting started

Your first steps with Play and your first 5 minutes of fun.

1. Play framework overview
2. Watch the screencast
3. Five cool things you can do with Play
4. Usability - details matter as much as features
5. Frequently Asked Questions
6. Installation guide
7. Your first application - the 'Hello World' tutorial
8. Set up your preferred IDE
9. Sample applications
10. Live coding script - to practice, and impress your colleagues with

Tutorial - Play guide, a real world app step-by-step

Learn Play by coding **'Yet Another Blog Engine'**, from start to finish. Each chapter will be a chance to learn one more cool Play feature.

1. Starting up the project
2. A first iteration of the data model
3. Building the first screen
4. The comments page
5. Setting up a Captcha
6. Add tagging support
7. A basic admin area using CRUD
8. Adding authentication
9. Creating a custom editor area
10. Completing the application tests
11. Preparing for production
12. Internationalisation and localisation

The essential documentation

Everything you need to know about Play.

1. Main concepts
    i. The MVC application model
    ii. A request life cycle
    iii. Application layout & organization
    iv. Development lifecycle
2. HTTP routing
    i. The routes file syntax
    ii. Routes priority
    iii. Serving static resources
    iv. Reverse routing: generate some URL
    v. Setting content types
    vi. Content negotiation
3. Controllers
    i. A Controller overview
    ii. Retrieve HTTP parameters
    iii. Advanced HTTP to Java binding
    iv. Return a result
    v. Interceptions
    vi. Session and Flash scopes
4. The template engine
    i. Template syntax
    ii. Template inheritance
    iii. Create tags
    iv. Extending the templates language
    v. Implicit objects
5. The domain object model
    i. Properties simulation
    ii. Setup a database to persist your model objects
    iii. Persist your object model with Hibernate
    iv. The stateless model
6. A little of asynchronism using jobs
    i. Bootstrap jobs
    ii. Scheduled jobs
    iii. Suspendable requests
7. Testing the application
    i. Writing tests
    ii. Fixtures
    iii. Running the tests
    iv. Continuous integration
8. Deployment options
    i. Standalone Play application

ii. Running on JEE application servers
iii. Google Application Engine
iv. Stax cloud hosting platform

Cookbook

Tips about everything and anything.

1. Using Play modules, and the modules repository
2. Security Guide
3. More about the JPA support
4. Using data validation
5. How to configure logs
6. Manage application.conf in several environments
7. Setting up i18n
8. Use cache and configure Memcached
9. Put your application in production
10. Sending e-mail
11. OpenID integration
12. Doing Ajax with JQuery

Distribution Modules

These optional modules are included with the standard distribution. More modules.

1. CRUD
2. Secure

References

Extended references for day-to-day hacking.

1. The missing cheat sheet
2. Browse API documentation
3. Template tags and extensions reference
4. Java extensions

Version notes

New versions of Play include certain changes.

1. Play 1.1
2. Play 1.0.3
3. Play 1.0.2
4. Play 1.0.1

The Play framework is a clean alternative to bloated Enterprise Java stacks. It focuses on developer productivity and targets RESTful architectures. Play is a perfect companion to **agile software development**.

The Play framework's goal is to ease web applications development while sticking with Java. Let's see how this is possible.

Wanna see some code?

Check Five cool things you can do with play, or start directly with the hello world tutorial.

A Java framework without the pain

Play is a pure Java framework and allows you to keep your preferred development tools and libraries. If you already use Java as a development platform you don't need to switch to another language, another IDE and other libraries. **Just switch to a more productive Java environment!**

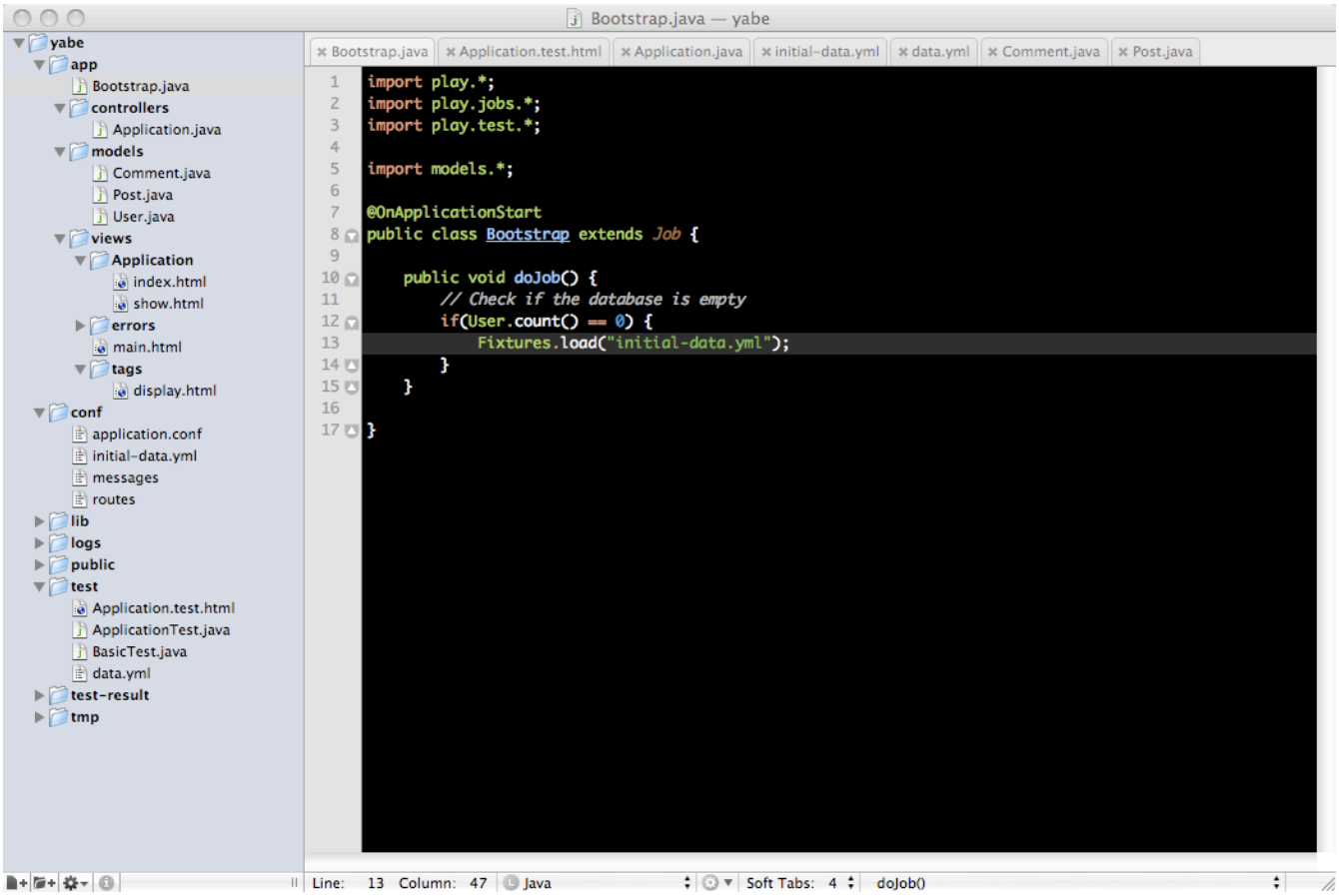
Fix the bug and hit Reload

The Java platform is infamous for its low productivity, mainly due to repeated and tedious compile-package-deploy cycles.

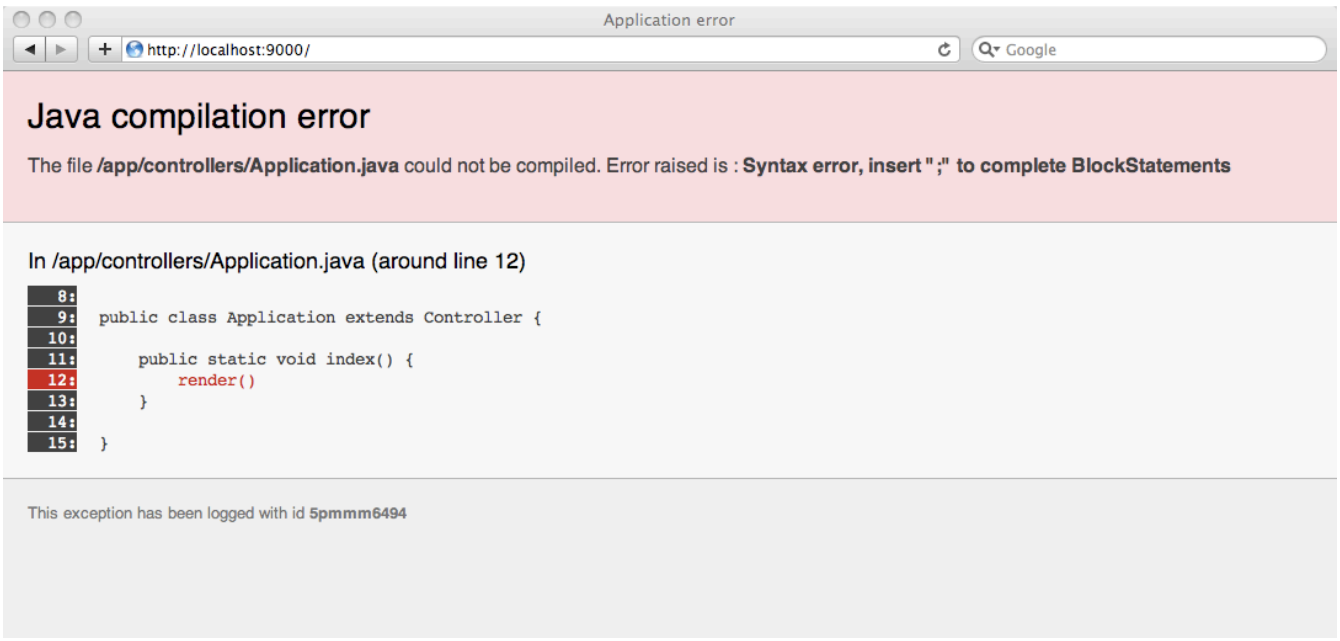
That's why we rethought the development cycle to make developing with Play an efficient process.

The framework compiles your Java sources directly and hot-reloads them into the JVM without the need to restart the server. You can then edit, reload and see your modifications immediately, just as in a LAMP or Rails environment.

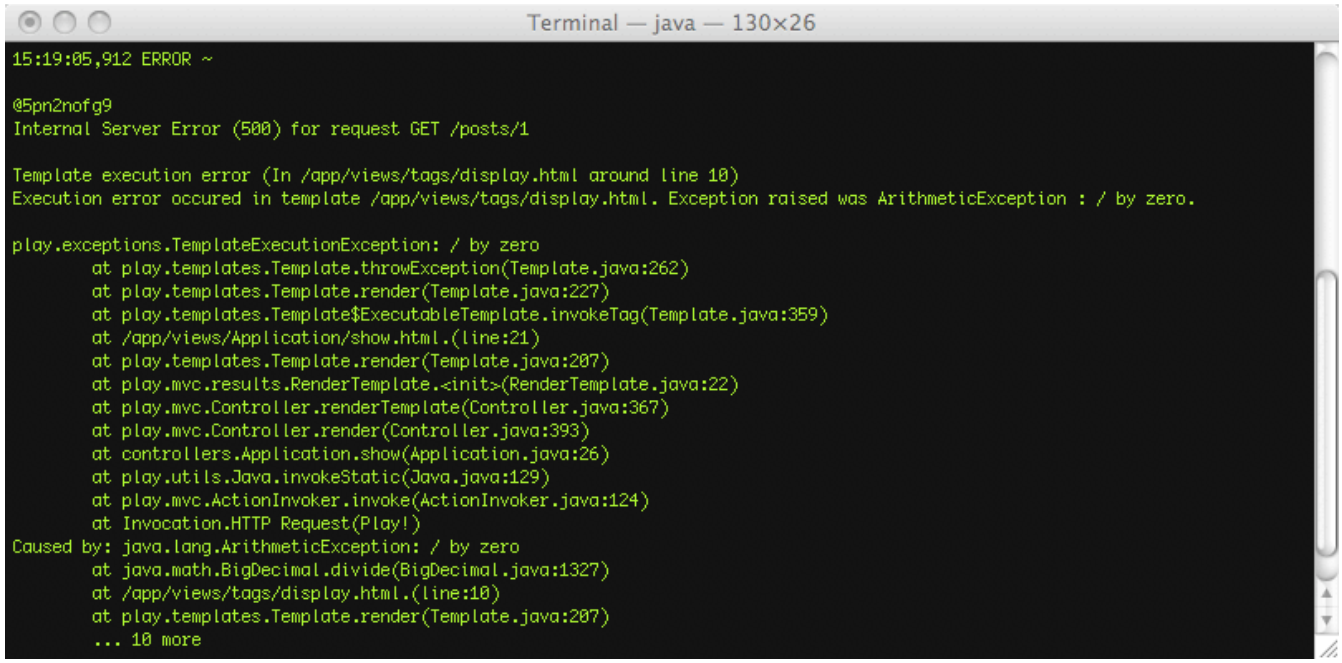
It's so fun that you can even work with a simple text editor and skip a full-featured Java IDE if you want.



Whenever an error occurs, the framework makes its best effort to identify and show you the problem.



Even the stack traces are stripped down and optimized to make it easier to solve problems. Look how template execution is nicely integrated to the Java stack trace.



Simple stateless MVC architecture

You've got a database on one side and a web browser on the other. Why should you have a state in between?

Stateful and component based Java Web frameworks make it easy to automatically save page state, but that brings a lot of other problems: what happens if the user opens a second window? What if the user hits the browser back button?

The Share Nothing architecture is promoted by many Web application frameworks from PHP to Ruby on Rails or Django. As the browser is becoming more and more powerful, it is now easy to use Ajax or offline storage to solve the state problems client-side.

We don't need to hack the HTTP model anymore to rebuild a fake state over the Web. Another aspect of 'share nothing' is to make it easier to render portions of the page in parallel, and to do partial page updates (or progressive enhancements).

HTTP-to-code mapping

If you're already using another Java Web framework like the Servlet API or the Struts framework, you have already used an abstract view of the HTTP protocol with strange Java APIs and concepts. We think differently. A Web application framework should give you full, straightforward access to HTTP and its concepts. This is a fundamental difference between Play and other Java web application frameworks.

HTTP, the Request/Response pattern, the REST architectural style, content-type negotiation, URI are all major concepts for the Play framework.

For instance, binding a URI pattern to a Java call is just one line:

```
GET     /clients/{id}         Clients.show
```

If Ajax, REST and managing back/forward movement between web pages are some of the issues you face in your day-to-day web development, just give Play a try.

Efficient templating engine

We like the idea behind JSP & Expression Language. But why do we need so many configuration files to create a tag library? Why can't we have full access to the underlying object model? JSP has a lot of limitations and this is indeed frustrating. That's why we've created a custom template system, inspired by JSP but without its constraints!

You and anyone else should be tired of writing things like this:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<c:choose>
    <c:when test="${emails.unread != null && fn:size(emails.unread)}">
        You have ${fn:size(emails.unread)} unread email(s)!
    </c:when>
    <c:otherwise>
        You have no unread emails!
    </c:otherwise>
</c:choose>
```

We think you would certainly prefer to write:

```
You have ${emails.unread ?: 'no'} ${emails.unread?.pluralize('email')} !
```

The expression language used by the play template engine is Groovy, which provides a syntax consistent with Java. While Play mainly uses the templating system to render HTML responses, you are free to use it to generate any other documents such as e-mail messages, JSON responses, etc.

JPA on steroids

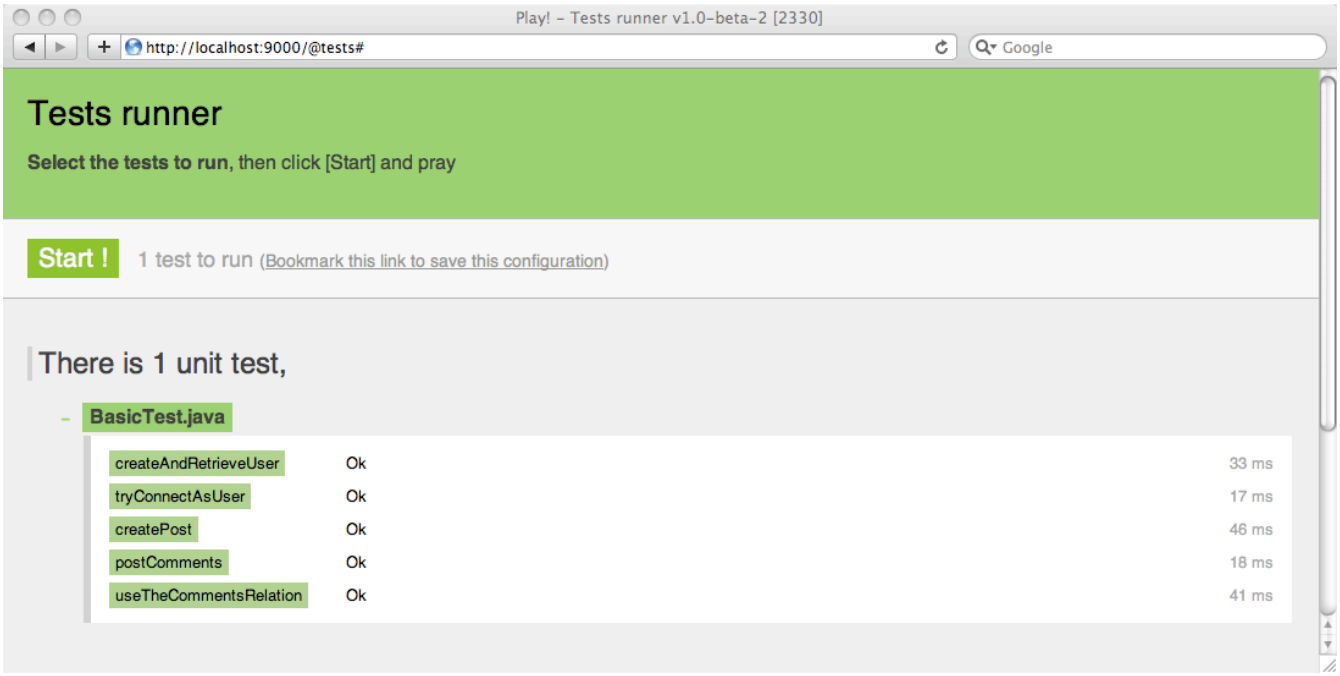
The Java Persistence API (JPA) is the cleanest object-relational mapping ORM) API available for Java. If you already know it you will be amazed how much simpler it becomes with Play. With nothing to configure, Play will automatically start the JPA Entity Manager and magically synchronize it while the code is reloaded.

Moreover if you use the provided **play.db.jpa.Model** superclass it will help make your code prettier. Have a look:

```
public void messages(int page) {
    User connectedUser = User.find("byEmail", connected());
    List<Message> messages = Message.find(
        "user = ? and read = false order by date desc",
        connectedUser
    ).from(page * 10).fetch(10);
    render(connectedUser, messages);
}
```

Test driven development (if you like it)

The integrated test runner makes it easy for you do test-driven development (TDD). You can write all kinds of tests, from simple unit tests to full acceptance tests, and run them directly in a browser using Selenium. Code coverage is also measured.



Full-stack application framework

The Play framework was initially inspired by our own Java applications. It has all the tools needed to create a modern web application, including:

- relational database support through JDBC.
- object-relational mapping using Hibernate (with the JPA API).
- integrated cache support, with easy use of the distributed memcached system if needed.
- straightforward web services consumption either in JSON or XML (we are talking real web services here; not the SOAP stuff).
- OpenID support for distributed authentication.
- your web application ready to be deployed anywhere (application server, Google App Engine, Cloud, etc…)
- image manipulation API.

The modular architecture lets you combine a web application with many others. Thanks to application modules, you can reuse your Java code, templates and static resources (such as JavaScript and CSS files) in a simple way.

Give it a try

Install the Play framework and start developing your first application.

Five examples to show the philosophy behind the Play framework.

Bind an HTTP parameter to a Java method parameter

Retrieving HTTP parameters from Java code is really simple with Play. Just declare method parameters with the same names as the HTTP parameters.

For example, with this request:

```
/articles/archive?date=08/01/08&page=2
```

You can retrieve the **date** and **page** parameters by declaring them as Java method parameters:

```
public static void archive(Date date, Integer page) {
    List<Article> articles = Articles.fromArchive(date, page);
    render(articles);
}
```

Play will use the method parameter's static type, to translate the HTTP value into a Java object.

Smart binding also works **with any class.**

```
public class Person {
    public String name;
    public Integer age;
}
```

A simplistic controller action to add a person may look like this:

```
public static void add(Person p) {
    p.save();
}
```

The HTML form defines fields with composite names:

```
<form action="/Directory/add" method="POST">
    Name: <input type="text" name="p.name" />
    Age: <input type="text" name="p.age" />
</form>
```

Redirect to an action by calling the corresponding Java method

There is no equivalent to the Java Servlet **forward** command with Play. But redirecting to another action is really simple. Just call the corresponding Java method and Play will generate the correct HTTP 'Redirect' response.

```
public static void show(Long id) {
    Article article = Article.findById(id);
    render(article);
}

public static void edit(Long id, String title) {
    Article article = Article.findById(id);
    article.title = title;
    article.save();
    show(id);
}
```

Note how at the end of the edit action, we redirect to the show action.

In any template you can use the equivalent syntax to generate the link:

```
<a href="@{Article.show(article.id)}">${article.title}</a>
```

That will generate the following HTML:

```
<a href="/articles/15">My new article</a>
```

Don't Repeat Yourself when passing Java objects to templates

In most Java frameworks, in order to pass Java objects to the template system you need to write something like:

```
Article article = Article.findById(id);
User user = User.getConnected();
Map<String, Object> model = new HashMap<String,Object>();
model.put("article", article);
model.put("user", user);
render(model);
```

With Play, you can just write:

```
Article article = Article.findById(id);
User user = User.getConnected();
render(article, user);
```

And retrieve objects from their Java local name in the template. That saves a lot of useless lines of code…

JPA on steroids

JPA is surely the best object-relational mapping (ORM) API available for Java. If you know it you will be amazed how much simpler it becomes with Play. With nothing to configure, Play will automatically start the JPA Entity Manager using Hibernate and magically synchronize it when code is reloaded.

Moreover, if you use the provided **play.db.jpa.Model** superclass it will help to make your code prettier. Take a look:

```
public void messages(int page) {
    User connectedUser = User.find("byEmail", connected());
    List<Message> messages = Message.find(
        "user = ? and read = false order by date desc",
        connectedUser
    ).from(page * 10).fetch(10);
    render(connectedUser, messages);
}
```

Straightforward file upload management

File upload management is very simple with Play.

The HTML form:

```
<form action="@{Article.uploadPhoto()}" method="POST"
enctype="multipart/form-data">
    <input type="text" name="title" />
    <input type="file" id="photo" name="photo" />
    <input type="submit" value="Send it..." />
</form>
```

And the Java code:

```
public static void uploadPhoto(String title, File photo) {
   ...
}
```

How could it be easier?

Perhaps the most striking thing about about the Play framework is that its biggest advantage over other Java web application development frameworks does not fit into a neat feature list, and is only apparent after you have used it to build something. That advantage is usability.

Note that usability is separate from functionality. In what follows, I am not suggesting that you cannot do this in some other framework: I merely claim that it is easier and more pleasant in Play I need to emphasise this because geeks often have a total blind spot for usability because they enjoying figuring out difficult things, and under-appreciate the value of things that Just Work.

Written by web developers for web developers

The first hint that something different is going on here is when you first hear that the Play framework is 'written by web developers for web developers', an unconventional positioning that puts the web's principles and conventions first and Java's second. Specifically, this means that the Play framework is more in line with the W3C's Architecture of the World Wide Web than it is with Java Enterprise Edition (Java EE) conventions.

URLs for perfectionists

For example, the Play framework, like other modern web frameworks, provides first-class support for arbitrary 'clean' URLs, which has always been lacking from the Servlet API. It is no coincidence that at the time of writing, Struts URLs for perfectionists, a set of work-arounds for the Servlet API-based Struts 1.x web framework, remains the third-most popular out of 160 articles on www.lunatech-research.com despite being a 2005 article about a previous-generation Java web technology.

In Servlet-based frameworks, the Servlet API does not provide useful URL-routing support; Servlet-based frameworks configure web.xml to forward all requests to a single controller Servlet, and then implement URL routing in the framework, with additional configuration. At this point, it does not matter whether the Servlet API was ever intended to solve the URL-routing problem and failed by not being powerful enough, or whether it was intended to be a lower-level API that you do not build web applications in directly. Either way, the result is the same: web frameworks add an additional layer on top of the Servlet API, itself a layer on top of HTTP.

Play combines the web framework, HTTP API and the HTTP server, which allows it to implement the same thing more directly with fewer layers and a single URL routing configuration. This configuration, like Groovy's and Cake PHP's, reflects the structure of an HTTP request - HTTP method, URL path, and then the mapping:

```
# Play 'routes' configuration file…

# Method   URL path         Controller

GET        /                Application.index
GET        /about           Application.about
POST       /item            Item.addItem
GET        /item/{id}       Item.getItem
GET        /item/{id}.pdf   Item.getItemPdf
```

In this example, there is more than one controller. We also see the use of an id URL parameter in the last two URLs.

Better usability is not just for normal people

Another way of looking at the idea that Play is by and for web developers is to consider how a web developer might approach software design differently to a Java EE developer. When you write software, what is the primary interface? If you are a web developer, the primary interface is a web-based user-interface constructed with HTML, CSS and (increasingly) JavaScript. A Java EE developer, on the other hand, may consider their primary interface to be a Java API, or perhaps a web services API, for use by other layers in the system.

This difference is a big deal, because a Java interface is intended for use by other programmers, while a web user-interface interface is intended for use by non-programmers. In both cases, good design includes usability, but

usability for normal people is not the same as usability for programmers. In a way, usability for everyone is a higher standard than usability for programmers, when it comes to software, because programmers can cope better with poor usability. This is a bit like the Good Grips kitchen utensils: although they were originally designed to have better usability for elderly people with arthritis, it turns out that making tools easier to hold is better for all users.

The Play framework is different because the usability that you want to achieve in your web application is present in the framework itself. For example, the web interface to things like the framework documentation and error messages shown in the browser is just more usable. Along similar lines, the server's console output avoids the pages full of irrelevant logging and pages of stack traces when there is an error, leaving more focused and more usable information for the web developer.

```
$ play run phase
~
~  _ __  | | __ _ _  _| |
~ | '_ \| |/ _' | || |_|
~ |  __/|_|\____|\__ (_)
~ |_|            |__/
~
~ play! 1.0, http://www.playframework.org
~
~ Ctrl+C to stop
~
Listening for transport dt_socket at address: 8000
10:15:58,629 INFO  ~ Starting /Users/peter/Documents/work/workspace/phase
10:16:00,007 WARN  ~ You're running Play! in DEV mode
10:16:00,424 INFO  ~ Listening for HTTP on port 9000 (Waiting a first request to start) ...
10:16:11,847 INFO  ~ Connected to jdbc:hsqldb:mem:playembed
10:16:13,448 INFO  ~ Application 'phase' is now started !
10:16:14,825 INFO  ~ starting DispatcherThread
10:16:48,168 ERROR ~

@61lagcl6i
Internal Server Error (500) for request GET /application/startprocess?account=x

Java exception (In /app/controllers/Application.java around line 41)
IllegalArgumentException occured : Person not found for account x

play.exceptions.JavaExecutionException: Person not found for account x
        at play.mvc.ActionInvoker.invoke(ActionInvoker.java:200)
        at Invocation.HTTP Request(Play!)
Caused by: java.lang.IllegalArgumentException: Person not found for account x
        at controllers.Application.startProcess(Application.java:41)
        at play.utils.Java.invokeStatic(Java.java:129)
        at play.mvc.ActionInvoker.invoke(ActionInvoker.java:127)
        ... 1 more
```

Try to imagine a JSF web application producing a stack trace this short. In fact, Play goes further: instead of showing the stack trace, the web application shows the last line of code within the application that appears in the stack trace. After all, what you really want to know is where things first went wrong in your own code.



This kind of usability does not happen by itself; the Play framework goes to considerable effort to filter out duplicate and irrelevant information, and focus on what is essential.

Quality is in the details

In the Play framework, much of the quality turns out to be in the details: they may be small things individually, rather than big important features, but they add up to result in a more comfortable and more productive development experience. The warm feeling you get when building something with Play is the absence of the frustration that usually results from fighting the framework.

By Peter Hilton, originally published on the Lunatech Research blog.

How does Play compare to framework X?

Nowadays, you have a very large number of choices for developing web applications. Play being a Java web application framework, let's compare it to other Java frameworks. Play is built for a 'share nothing' architecture (think about it as a stateless framework), so it's very different from all these Java stateful and components based frameworks like Seam, Tapestry or Wicket. It's closer to Spring MVC or Struts (or Struts 2), but way more opinionated.

However Play is a unique Java framework. It does not really rely on the so-called Java Enterprise standards. It uses Java, but tries to bring all the good things from the frameworks based on scripting languages like Ruby On Rails, Django, Pylons, Symfony, Grails and Cake PHP to the Java world. We really tried to get the best out of the Java platform without getting the pain of traditional Java web development: a slow development cycle, too much abstraction and too much configuration.

Why don't you rename the 'play' package to 'org.playframework'?!

You are missing the point. Play is not another library that you add to your Servlet container. It's really a full stack Java framework that runs your application stand-alone. The Java package naming conventions exist to avoid name clashes when you use several different libraries from several vendors. But believe us, you will never put the **play.jar** library in your own project. This is not the way Play works. Play **is the platform**. Don't worry about that.

Why do I need Python (I would prefer Maven)?

We need a lot of scripts to manage Play applications, for creating a new application, running it, launching a browser, etc… Of course we could write these in Java directly. But running a Java VM for simple tasks like project creation is very slow. And Java itself is very limited when it comes to OS interaction.

Python allowed us to write these scripts in a completely portable way. It's fast to run, easy to write and available out of the box on most systems. It's not on Windows, that's why we bundle a Windows Python binary with the framework.

Is Play a Groovy framework?

No. Even though we use Groovy as the base technology for the Play templating system, it's totally transparent. Also, you can't directly write any other part of the application (such as controllers, models or other utilities) in Groovy. If you're looking for a Groovy based framework you should have a look at Grails.

You guys don't even know how to program in Java...

We are fully aware that we made choices that are pretty uncommon in the Java world, and that Play does not blindly follow all the so-called Java 'good practices'. But all of the Play team members are very experienced Java developers and we are totally aware of the choices we made and the rules we broke.

Java itself is a very generic programming language and not originally designed for web application development. It is a very different thing to write a generic and reusable Java library and to create a web application. A web application itself doesn't need to be designed to be reusable. You need less abstraction, less configuration. Reusability does exist for web applications, but through web service APIs rather than language-level integration.

When the development time tends to zero you can concentrate on your application features and experiment quickly, instead of trying to abstract things for future developments.

Is Play fast?

Yes, Play itself is fast. But that doesn't mean that any particular application will be fast. The embedded HTTP server, the basic routing and the controller invocation stack are very very fast. Used with a modern JVM and Just In Time compiling you can easily get thousands of requests per second. Unfortunately, your application will likely use a database, which will become the bottleneck as usual.

The biggest CPU consumer in the Play stack at the moment is the Groovy-based template engine. But as Play applications are easily scalable, it is not really a problem if you need to handle very high traffic: you can balance the load between several servers. We also hope for a performance gain at this level with the new JDK7 and its better support for dynamic languages.

Can I already use Play for a production application?

Sure, a lot of people already use Play in production. The 1.0 branch is now in maintenance mode, which means that we will just fix bugs and keep API compatibility in that branch.

Is library X supported?

Play is standard Java, so any standard Java library can easily be used. However keep in mind that Play does not use the Servlet API (although you can get it work in a standard Servlet container using the WAR export feature). So unless the library you want to use relies on the Servlet API, there won't be any problem.

How can I help/contribute?

We use launchpad as our main development tool, and launchpad itself is very open. You can just register a launchpad account and start to contribute. You can assign any open bug to yourself, fork any Play official branch to your own branch, and propose a merge to us if you have fixed something.

The documentation itself is kept as Textile files in the framework source repository, so you can edit it and contribute just as you do with code.

Prerequisites

To run the Play framework, you need Java 5 or later. If you wish to build Play from source, you will need the Bazaar source control client to fetch the source code and Ant to build it.

> If you are using MacOS, Java is just built-in. If you are using Linux, make sure to use either the Sun-JDK or OpenJDK (and not gcj which is the default Java command on many Linux distros). If you are using Windows, just download and install the latest JDK package.

Be sure to have Java in the current path (enter **java -version** to check). Play will use the default Java or the one available at the **$JAVA_HOME** path if defined.

The **play** command line utility uses Python. So it should work out of the box on any UNIX system (however it requires at least Python 2.5). If you're running Windows, don't worry, a Python runtime is bundled with the framework.

Download the binary package

Download the latest Play binary package and extract the archive. For convenience, you should add the framework installation directory to your system PATH. If you're on UNIX, make sure that the **play** script is runnable (otherwise simply do a **chmod +x play**). That's all.

> **Tip**
>
> If you need to rebuild the framework for some reason, run **ant** in the **$PLAY_HOME/framework/** directory.

Build from the latest sources

To benefit from the latest improvements and bug fixes, you may want to compile Play from sources. You'll need a Bazaar client to fetch the sources and Ant to build the framework.

From the command line:

```
# bzr checkout lp:play
# cd play/framework
# ant
```

The Play framework is ready to use.

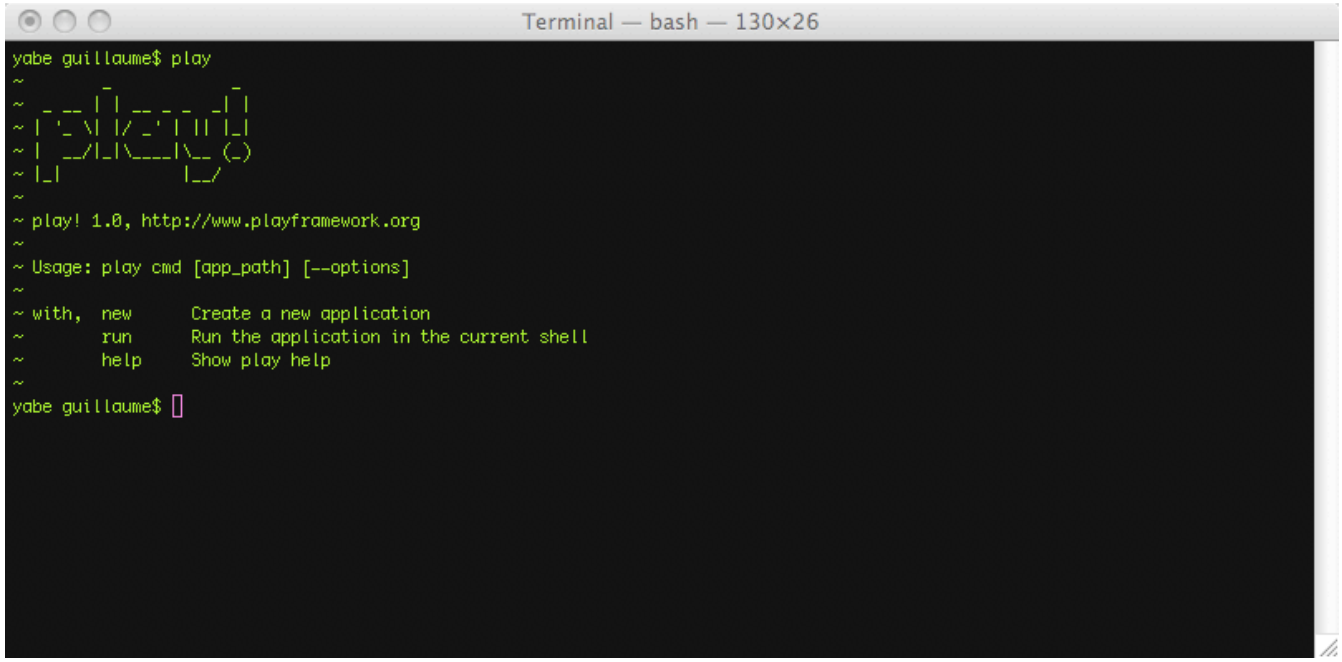
> **lp:play** currently aliases to **http:bazaar.launchpad.net/%7Eplay-developers/play/1.0/**

Using the play command

When the framework is correctly installed, open a shell and execute **play**.

```
$ play
```

You should see the play default message:



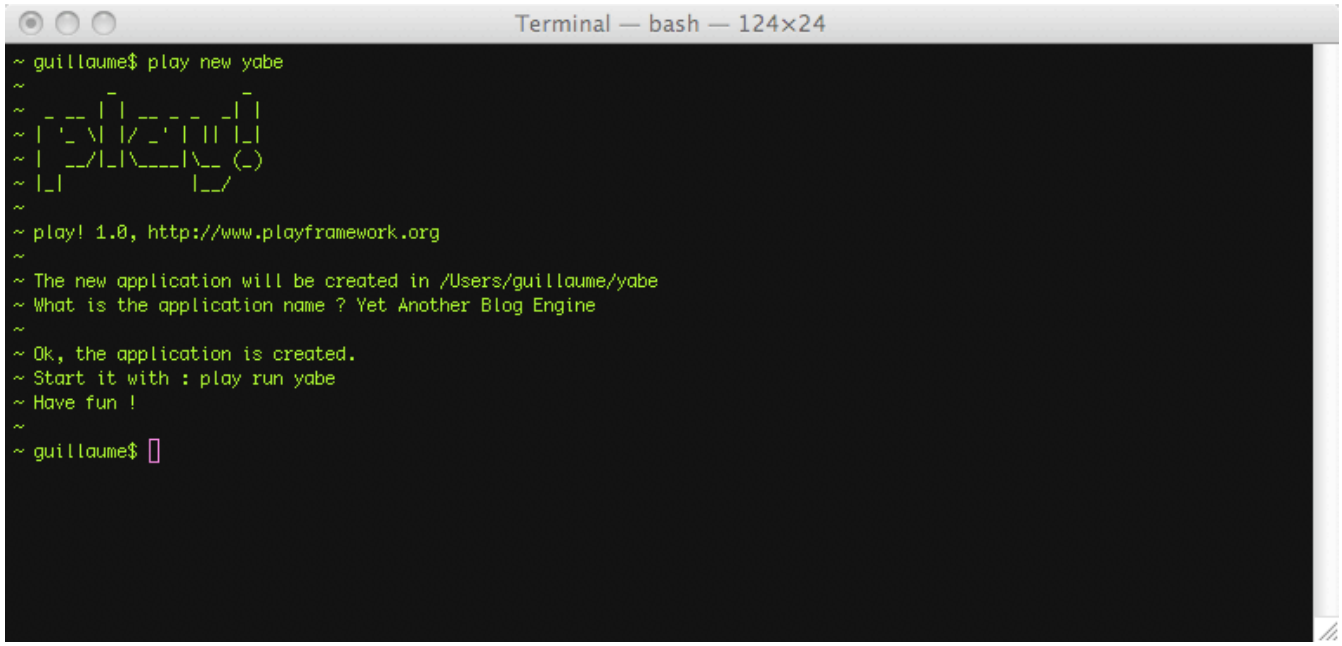
You can get more help on any specific command using **play help any-command-name-here**. For example, try:

```
# play help run
```

Creating a new application

Use the **new** command to create a new application. You must give a non-existent path where the application will be created.

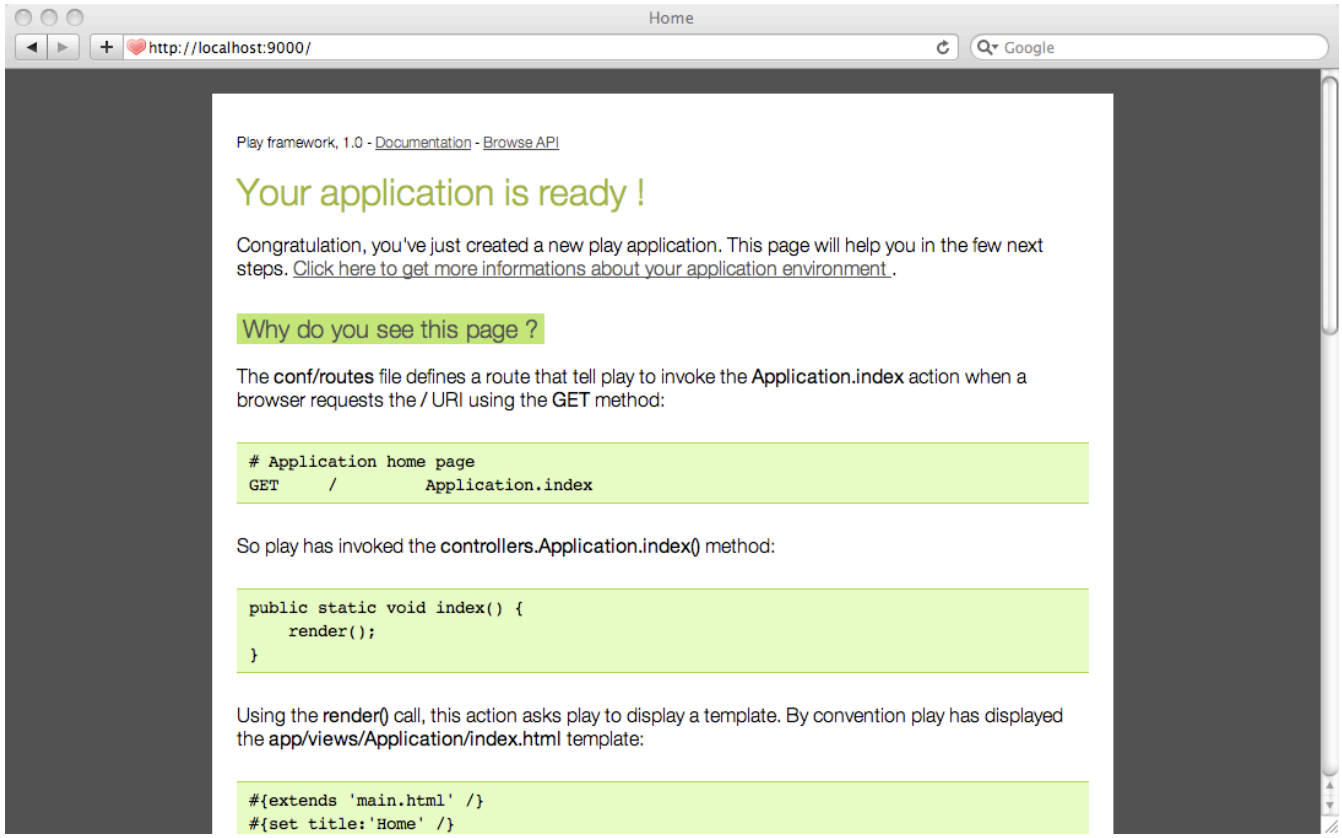
```
# play new myApp
```



Will create a new application. Start it with:

```
# play run myApp
```

You can then open a browser at http://localhost:9000 and see the default page for the application.



Your Play environnement is ready

See how to set-up your preferred IDE.

A very simple tutorial, that will just help you to discover the Play framework with the famous 'Hello World' example.

Prerequisites

First of all, make sure that you have a working Java installation. Play requires **Java 5 or later**.

As we will use the command line a lot, it's better to use a Unix-like OS. If you run a Windows system, it will also work fine; you'll just have to type a few commands in the command prompt.

You will of course need a text editor. If you are accustomed to use a full featured Java IDE like Eclipse or Netbeans you can of course use it. However with Play you can have fun working with a simple text editor like Textmate, Emacs or VI. This is because the framework manages the compilation and the deployment process itself. We will soon see that…

Installing the Play framework

Installation is very simple. Just download the latest binary package from the download page and unzip it to any path.

> If you're using Windows, it is generally a good idea to avoid space characters in the path, so for example **c:\Play** would be a better choice than **c:\Documents And Settings\user\play**.

To work efficiently, you need to add the Play directory to your working path. It allows to type just **play** at the command prompt to use the Play utility. To check that the installation worked, just open a new command line and type **play**; it should show you the Play basic usage help.

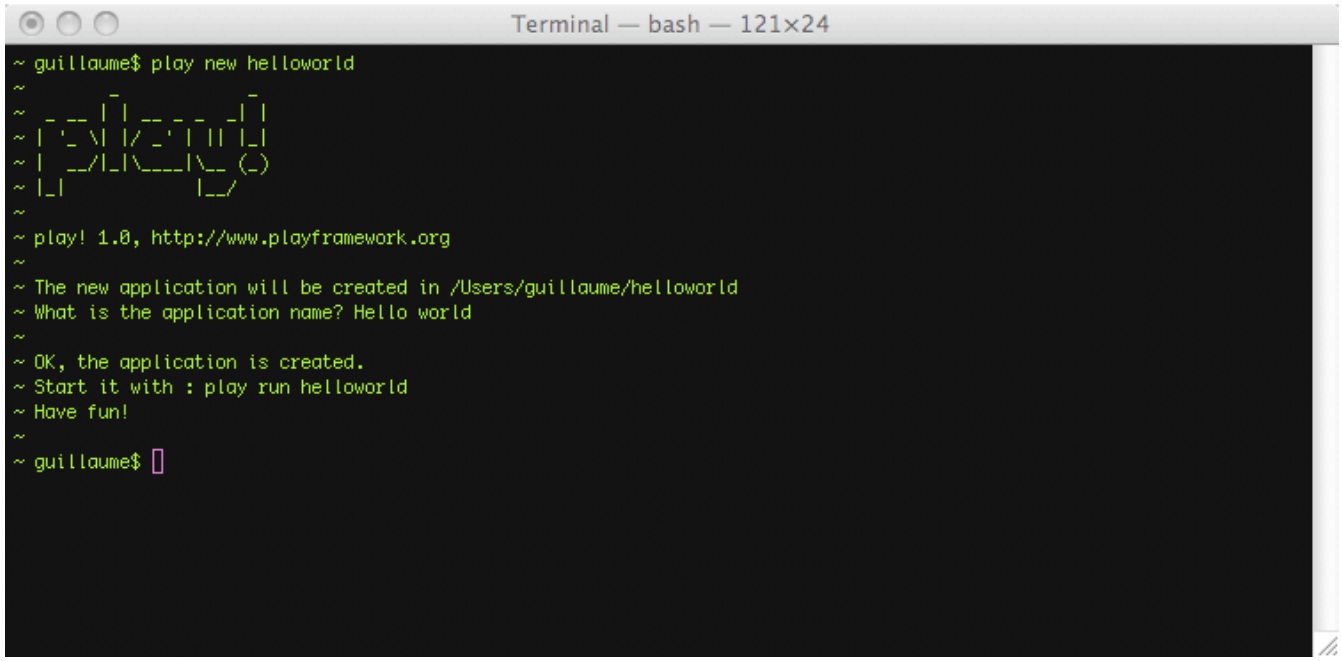
Project creation

Now that Play is correctly installed, it's time to create the hello world application. Creating a Play application is pretty easy and fully managed by the Play command line utility. That allows for standard project layouts between all Play applications.

Open a new command line and type:

```
~$ play new helloworld
```

It will prompt you for the application full name. Type **Hello world**.



The **play new** command creates a new directory **helloworld/** and populates it with a series of files and directories, the most important being:

app/ contains the application's core, split between models, controllers and views directories. It can contain other Java package as well. This is the directory where **.java** source files live.

conf/ contains all the application's configuration files, especially the main **application.conf** file, the **routes** definition files and the **messages** files used for internationalization.

lib/ contains all optional Java libraries packaged as standard .jar files.

public/ contains all the publicly available resources, which includes JavaScript, stylesheets and images directories.

test/ contains all the application tests. Tests are either written either as Java JUnit tests or as Selenium tests.

> Because **Play uses UTF-8** as single encoding, it's very important that all text files hosted in these directories are encoded using this charset. Make sure to configure your text editor accordingly.

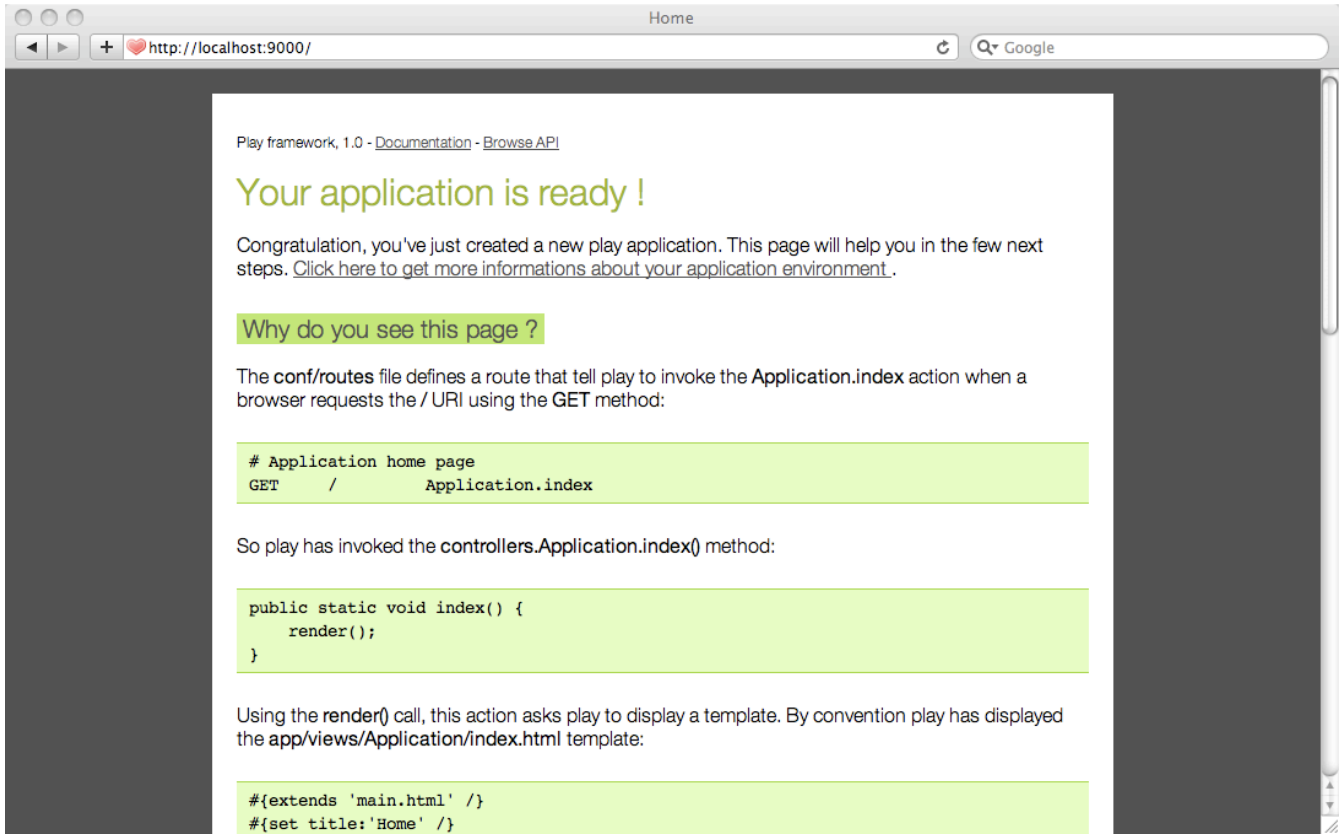
Now if you're a seasoned Java developer, you may wonder where all the .class files go. The answer is nowhere: Play doesn't use any class files but reads the Java source files directly. Under the hood we use the Eclipse compiler to compile Java sources on the fly.

That allows two very important things in the development process. The first one is that Play will detect changes you make to any Java source file and automatically reload them at runtime. The second is that when a Java exception occurs, Play will create better error reports showing you the exact source code.

Running the application

We can now test the newly-created application. Just return to the command line, go to the newly-created **helloworld/** directory and type **play run**. Play will now load the application and start a Web server on port 9000.

You can see the new application by opening a browser to http://localhost:9000. A new application has a standard welcome page that just tells you that it was successfully created.



Let's see how the new application can display this page.

The main entry point of your application is the **conf/routes** file. This file defines all of the application's accessible URLs. If you open the generated routes file you will see this first 'route':

```
GET          /                          Application.index
```

That simply tells Play that when the web server receives a **GET** request for the **/** path, it must call the **Application.index** Java method. In this case, **Application.index** is a shortcut for **controllers.Application.index**, because the controllers package is implicit.

When you create standalone Java applications you generally use a single entry point defined by a method such as:

```
public static void main(String[] args) {
  ...
}
```

A Play application has several entry points, one for each URL. We call these methods **action** methods. Action methods are defined in special classes that we call **controllers**.

Let's see how the **controllers.Application** controller looks like. Open the **helloworld/app/controllers/Application.java** source file:

```
package controllers;

import play.mvc.*;

public class Application extends Controller {

        public static void index() {
                render();
        }

}
```

You see that controller classes extend the **play.mvc.Controller** class. This class provides all useful methods for controllers, like the **render()** method we use in the index action.

The index action is defined as a **public static void** method. This is how action methods are defined. You can see that action methods are static, because the controller classes are never instantiated. They are marked public to authorize the framework to call them in response of a URL. They always return void.

The default index action is simple: it calls the **render()** method which tells Play to render a template. Using a template is the most common way (but not the only one) to generate the HTTP response.

Templates are simple text files that live in the **/app/views** directory. Because we didn't specify a template, the default one for this action will be used: **Application/index.html**

To see what the template looks like, open the **helloworld/app/views/Application/index.html** file:

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{welcome /}
```

The template content seems pretty light. In fact, all you see are Play tags. Play tags are very close to JSP taglib. This is the **#{welcome /}** tag that generates the welcome message you've seen in the browser.

The **#{extends /}** tags tells Play that this template inherits another template called **main.html**. Template inheritance is a powerful concept that allows to create complex web pages by reusing common parts.

Open the **helloworld/app/views/main.html** template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        <link rel="stylesheet" type="text/css" media="screen"
            href="@{'/public/stylesheets/main.css'}" />
        <link rel="shortcut icon" type="image/png"

href="@{'/public/file:///Users/nicolasleroux/Projects/Personal/1.1/documentation/images/
/>
    </head>
    <body>
        #{doLayout /}
    </body>
</html>
```

Can you see the **#{doLayout /}** tag? This is where the content of **Application/index.html** will be inserted.

Creating the form

We will start the 'Hello World' application with a form where you can enter your name.

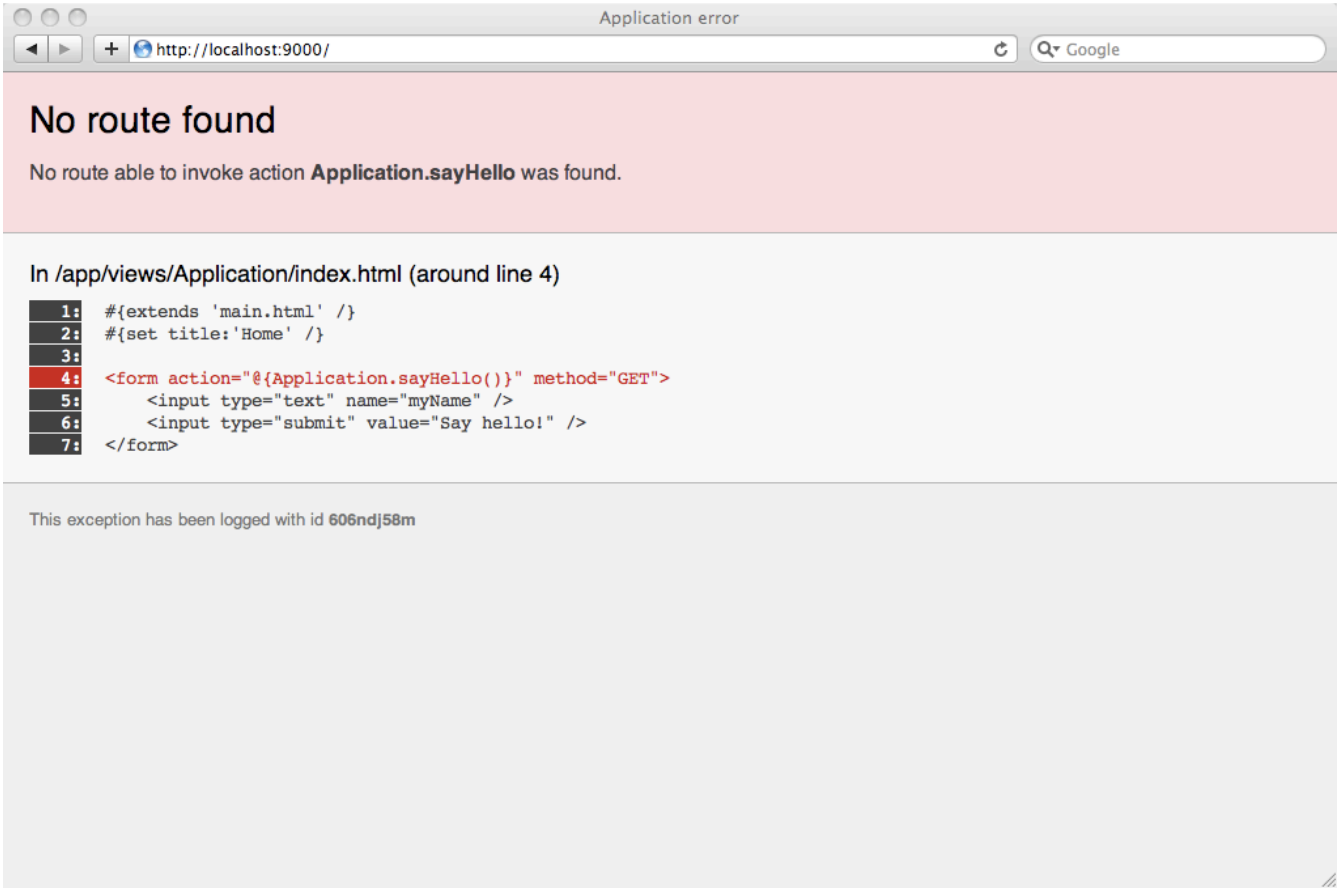
Edit the **helloworld/app/views/Application/index.html** template:

```
#{extends 'main.html' /}
#{set title:'Home' /}

<form action="@{Application.sayHello()}" method="GET">
    <input type="text" name="myName" />
    <input type="submit" value="Say hello!" />
</form>
```

Note that we use GET as form submission method, because the form submission does not have any side effect and is idempotent.

We use the **@{...}** notation to ask Play to automatically generate the URL able to invoke the **Application.sayHello** action. Now, refresh the home page in the browser.



Oops, you get an error. This is because you reference the non-existent action **Application.sayHello**. Let's create it in the **helloworld/app/controllers/Application.java** file:

```
package controllers;

import play.mvc.*;

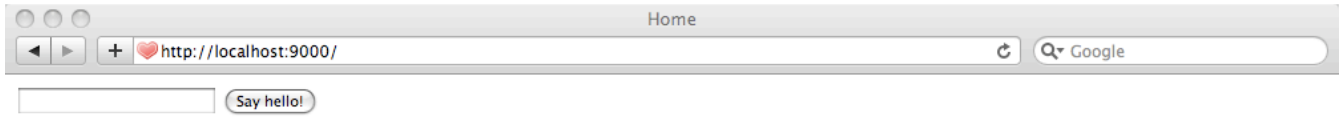
public class Application extends Controller {

    public static void index() {
        render();
    }

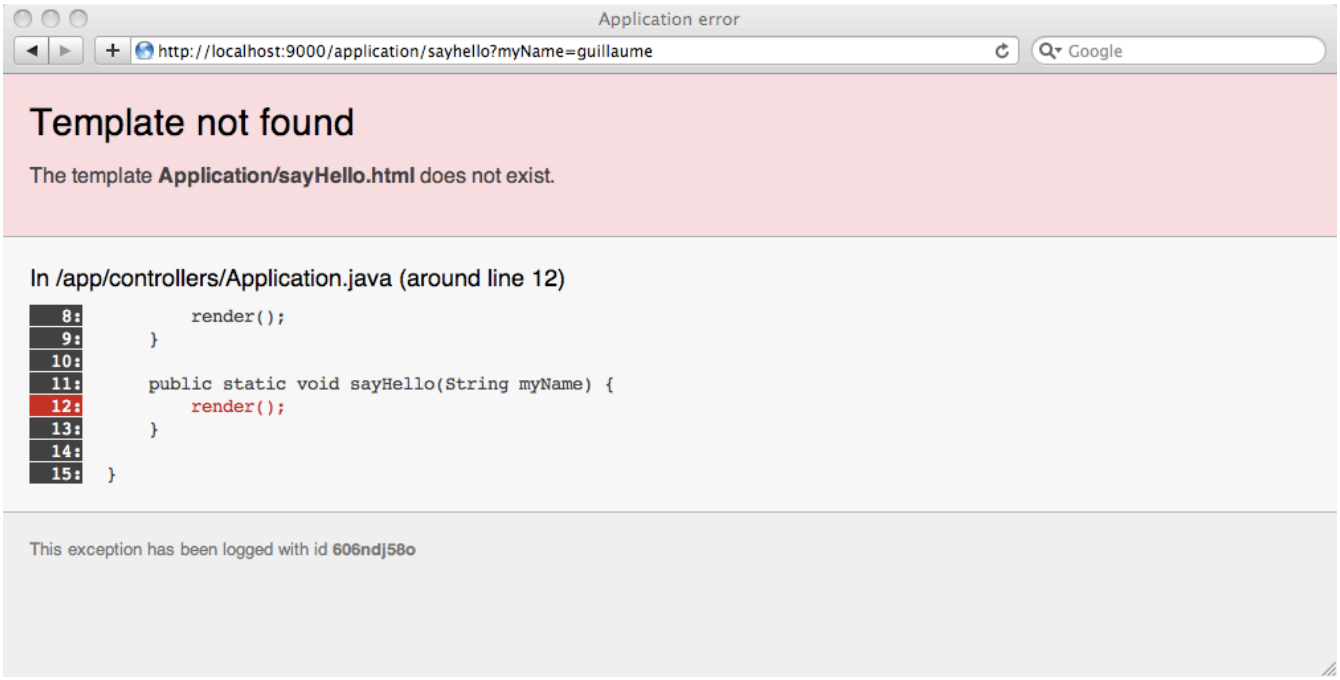
    public static void sayHello(String myName) {
        render(myName);
    }

}
```

We have declared the **myName** parameter in the action method signature, so it will automatically be filled with the value of the HTTP **myName** parameter, coming from the form submission. And we call render to just display a template; as we pass the **myName** variable to the **render()** call, this one will be available from the template.



Now, if you try to enter your name and submit the form, you will get another error:



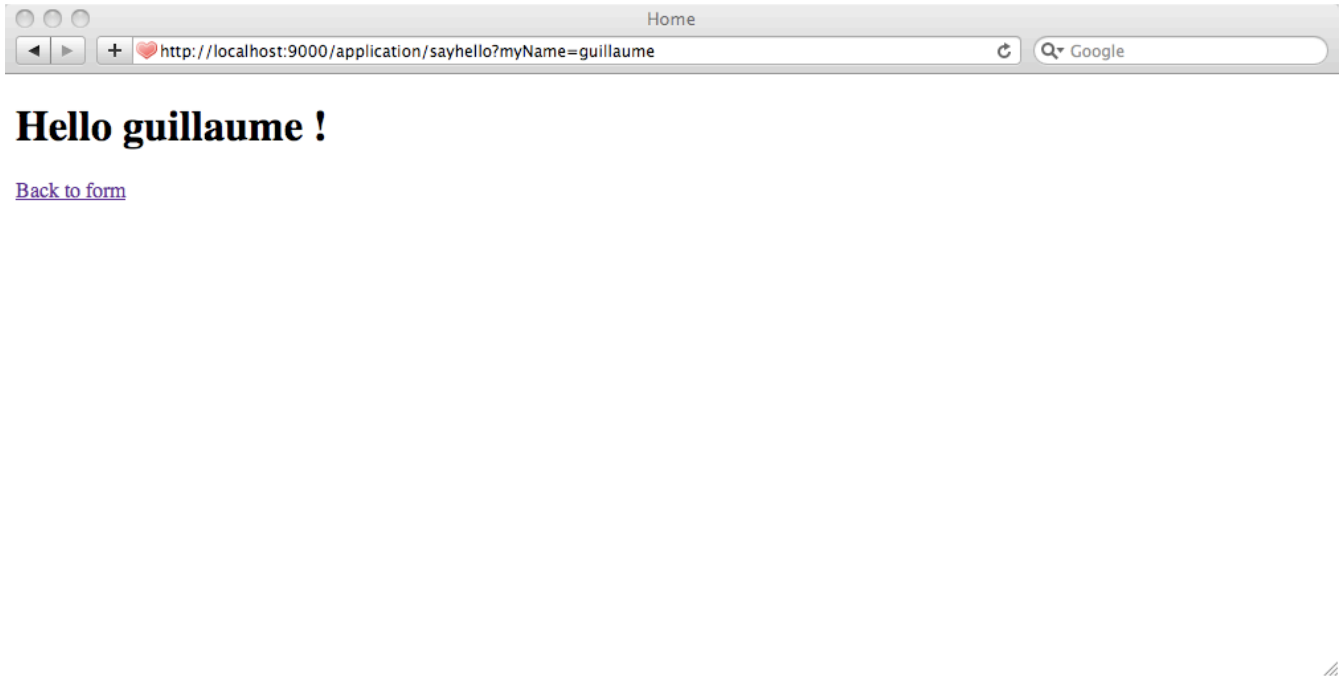
The error is pretty clear. Play tries to render the default template for this action method, but it doesn't exist. Let's create it in the file **helloworld/app/views/Application/sayHello.html**:

```
#{extends 'main.html' /}
#{set title:'Home' /}

<h1>Hello ${myName ?: 'guest'}!</h1>

<a href="@{Application.index()}">Back to form</a>
```

You can refresh the page.



Look how we have used Groovy's **?:** operator. It switches to a default value if the **myName** variable is not filled. So if you try to submit the form without entering any name, it will display 'Hello guest'.

Providing a better URL

Well, if you look at the submission URL, you will see something like:

```
http://localhost:9000/application/sayhello?myName=guillaume
```

It is not very pretty. This is because Play used the default 'catch all' route.

```
*       /{controller}/{action}                  {controller}.{action}
```

We can have a better URL by specifying a custom path for the **Application.sayHello** action. Edit the **helloworld/conf/routes** file and add this route after the first one:

```
GET     /hello                                  Application.sayHello
```

Now go back to the form and submit it again to check that it uses the new URL pattern.

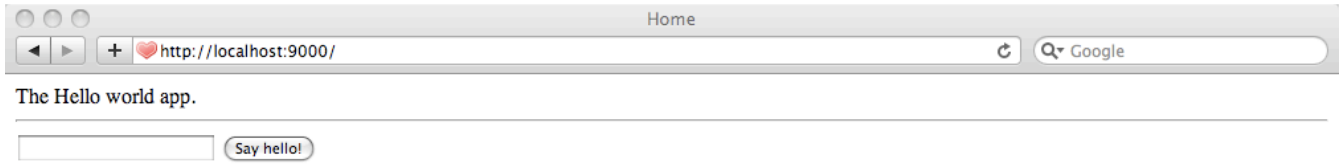
Customizing the layout

As both templates that the application use so far inherit from the same **main.html** template, you can easily add a custom layout. Edit the **helloworld/app/views/main.html** file:

```
...
<body>
    The Hello world app.
    <hr/>

    #{doLayout /}
</body>
...
```

Now we have a common header for both pages.



Adding validation

We will add a little validation to the form, to make the name field required. We can use the Play validation framework to do that.

Edit the **helloworld/app/controllers/Application.java** controller, and the **sayHello** action:

```
...
public static void sayHello(@Required String myName) {
    if(validation.hasErrors()) {
        flash.error("Oops, please enter your name!");
        index();
    }
    render(myName);
}
...
```

And don't forget to import the **play.data.validation.*** package to get the **@Required** annotation. Play will automatically check that the myName field is filled or will add an error object to the **errors** scope. Then if there is any error, we add a message to the **flash** scope and redirect to the **index** action.

The flash scope allows to keep messages during action redirection.

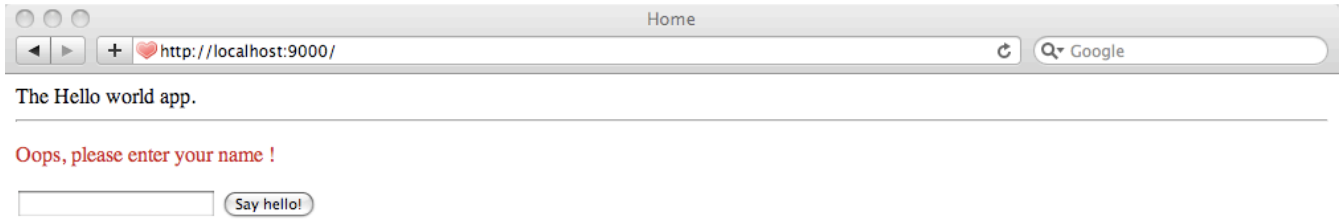
Now you just have to display the **error** message if any. Edit the **helloworld/app/views/Application/index.html**:

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{if flash.error}
    <p style="color:#c00">
        ${flash.error}
    </p>
#{/if}

<form action="@{Application.sayHello()}" method="GET">
    <input type="text" name="myName" />
    <input type="submit" value="Say hello!" />
</form>
```

And check that it works:



Writing an automated test suite

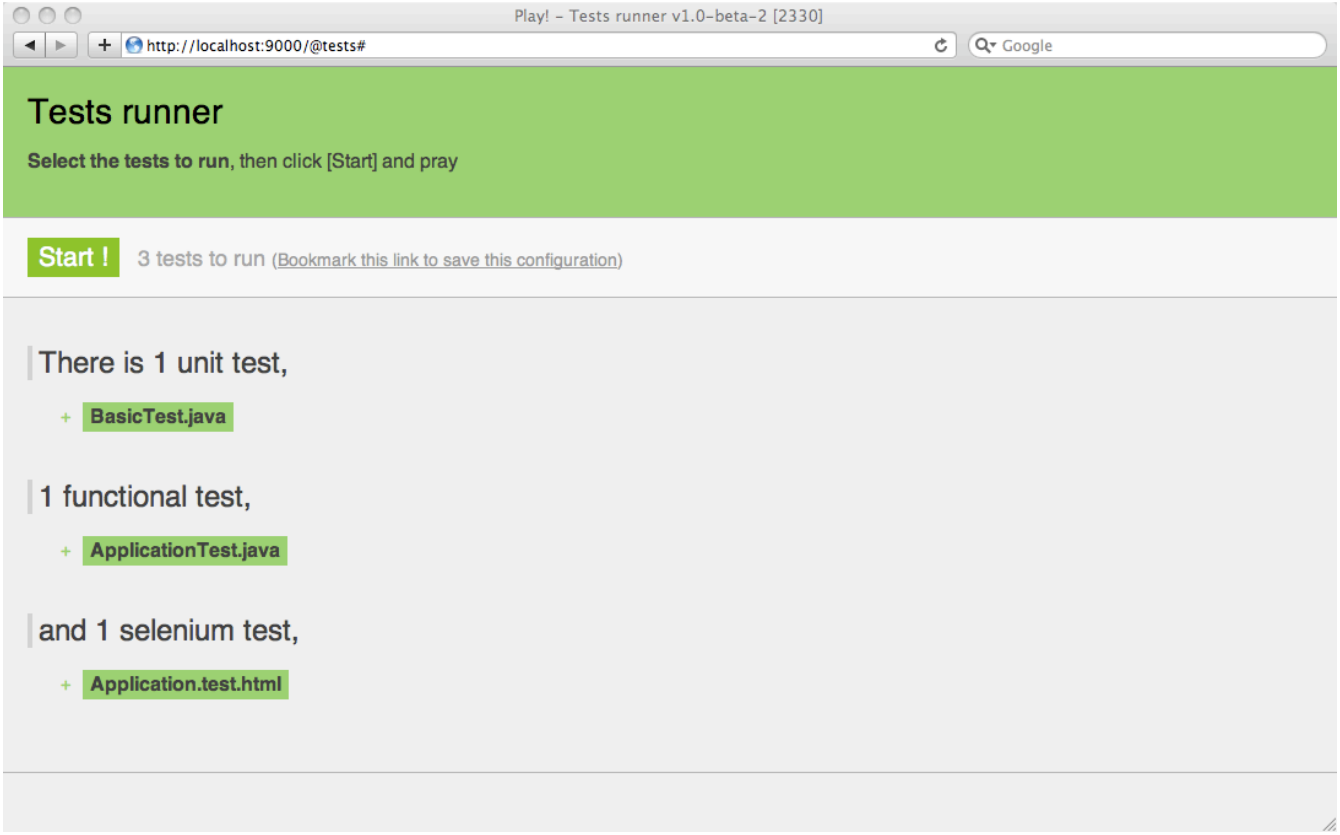
We will finish by writing a test for the application. As there is no Java logic to test, we need to test the Web application itself. So we will write a Selenium test.

First, you need to run your application in **test mode**. Stop the application and restart it with the **test** command:

```
$ play test
```

The **play test** command is almost the same as **play run**, except that it loads a test runner module that allows to run test suite directly from a browser.

Open a browser to the http://localhost:9000/@tests URL to see the test runner. Try to select all the default tests and run them; all should be green… But these default tests don't really test anything.



A selenium test suite is typically written as an HTML file. The HTML syntax required by selenium is a little tedious (formatted using an HTML table element) to write. The good news is that Play will help you to generate it using the Play template engine and a set of tags that support a simplified syntax for selenium scenarios.

The default test suite of a newly created Play application already contains a selenium test. Open the **helloworld/test/Application.test.html** file:

```
*{ You can use plain selenium command using the selenium tag }*

#{selenium}
    // Open the home page, and check that no error occurred
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
#{/selenium}
```

This test should run without any problem with the application for now. It just open the home page and check that the page content does not contain the 'Application error' text.

Let's write a test for our application. Edit the test content:

```
#{selenium}
    // Open the home page, and check that no error occurred
    open('/')
    assertNotTitle('Application error')

    // Check that it is the form
    assertTextPresent('The Hello world app.')

    // Submit the form
    clickAndWait('css=input[type=submit]')

    // Check the error
    assertTextPresent('Oops, please enter your name!')

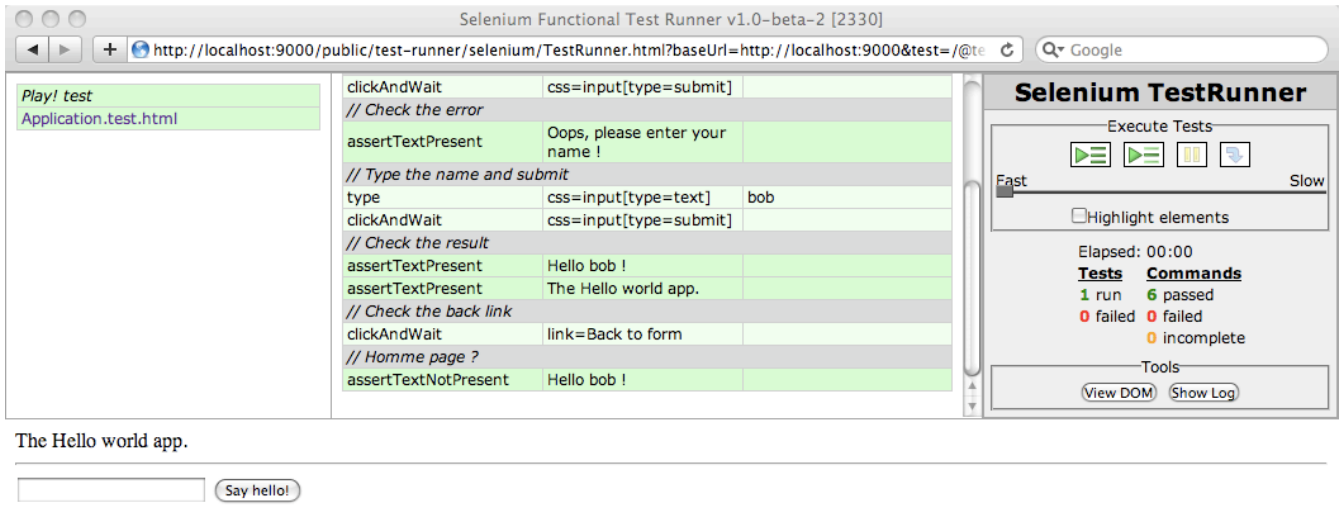
    // Type the name and submit
    type('css=input[type=text]', 'bob')
    clickAndWait('css=input[type=submit]')

    // Check the result
    assertTextPresent('Hello bob!')
    assertTextPresent('The Hello world app.')

    // Check the back link
    clickAndWait('link=Back to form')

    // Home page?
    assertTextNotPresent('Hello bob!')
#{/selenium}
```

We have fully tested the application. Just select this test in the test runner and click 'Start'. Everything should be green!



Learn more

This was a very simple tutorial about a very dumb application. If you want to learn more about Play, you can now check out the Play guide.

Working with Play is easy. You don't even need a sophisticated IDE as Play compiles and refreshes the modifications you make to your source files automatically. So you can easily work using a simple text editor.

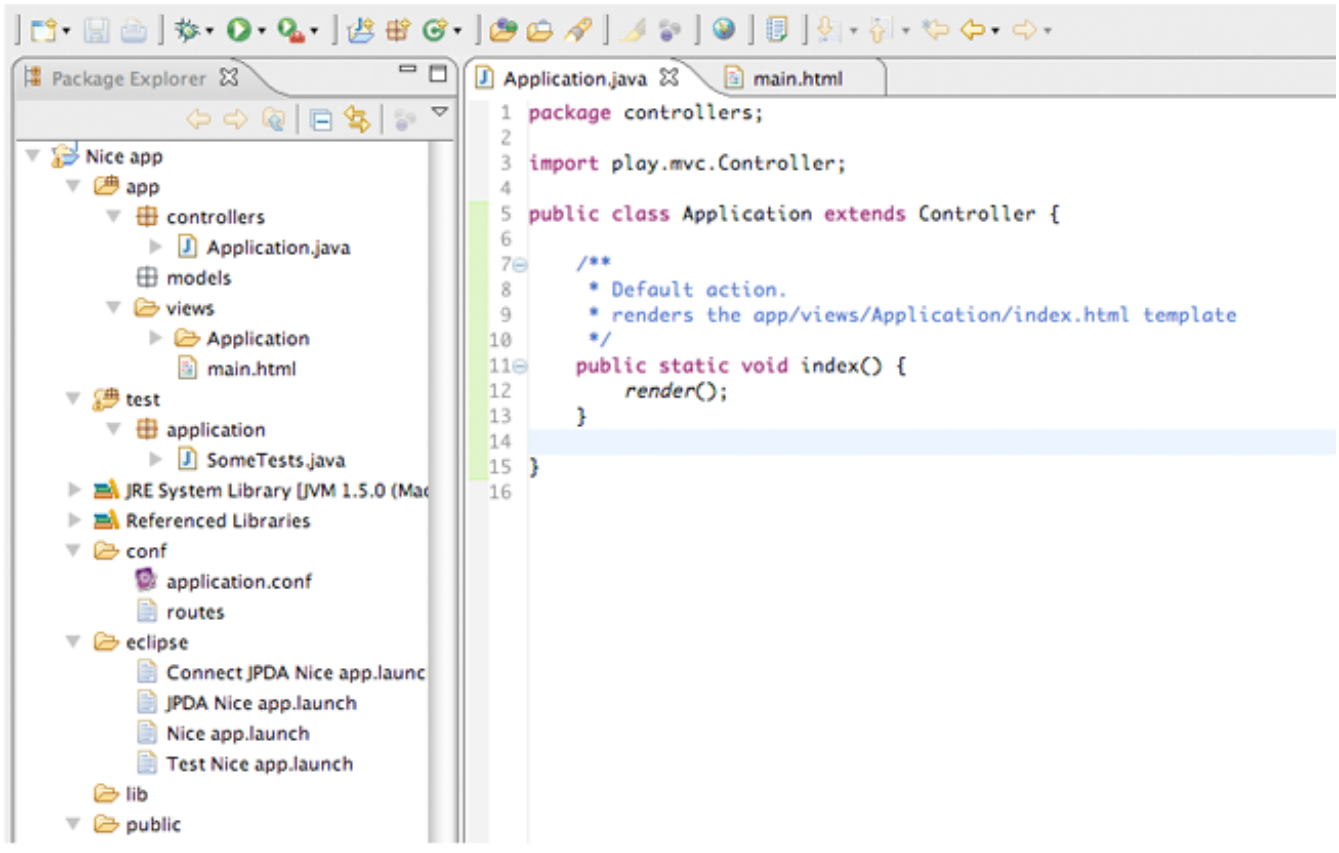
However, using a modern Java IDE provides cool productivity features like auto-completion, on-the-fly compilation, assisted refactoring and debugging. Play supports the NetBeans, IntelliJ IDEA and Eclipse platforms.

Generate configuration files for Eclipse

Play provides a command to simplify Eclipse configuration. To transform a Play application into a working Eclipse project, use the **eclipsify** command:

```
# play eclipsify myApp
```

You then need to import the application into your Workspace with the **File/Import/General/Existing project…** menu.



The **eclipsify** command generates several launchers for the application. The main launcher in only usable with the **Run As** Eclipse command. You can then use the **Connect JPDA launcher** using **Debug As** to start a debugging session at any time. Stopping the debugging session will not stop the server.

If you make any important changes to your application, such as changing the classpath, use **eclipsify** again to regenerate the configuration files.

Additionally, an Eclipse plugin is available in your Play distribution, in the **support/eclipse/** directory. To install it, simply copy the JAR file you will find to your Eclipse installation's **dropins** folder.

Do not commit Eclipse configuration files when you work in a team!

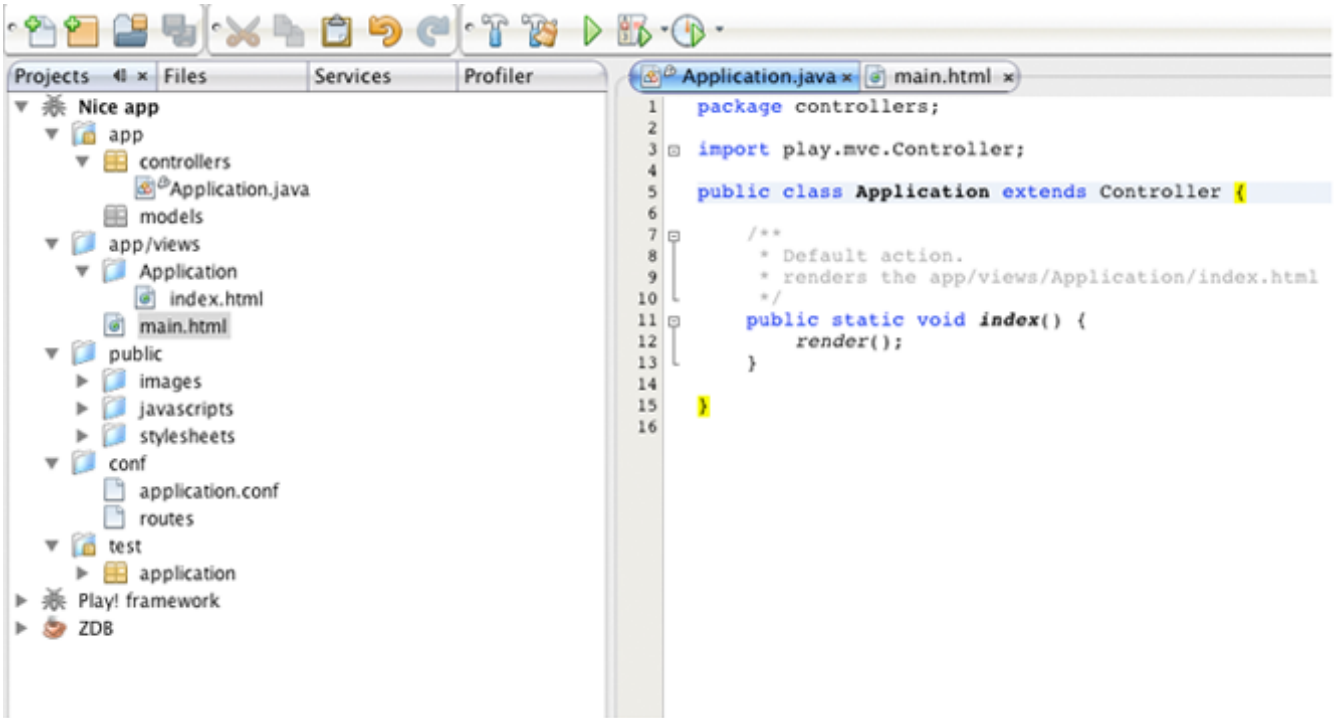
The generated configuration files contain absolute references to your framework installation. These are specific to your own installation. When you work in a team, each developer must keep his Eclipse configuration files private.

Generate configuration files for NetBeans

Play provides a command to simplify NetBeans configuration. To transform an existing application to a valid NetBeans project, use the **netbeansify** command:

```
# play netbeansify myApp
```

Then you can just open the application as a NetBeans project.



Use the standard **Run** button to start the application. When the application is started you can attach a debugging session at any time using the **Debug** button. Stopping the debugging session doesn't stop the server.

If you make any important change to your application such as changing the classpath, use **netbeansify** again to regenerate the configuration files.

Do not commit the nbproject/ directory when you work in a team!

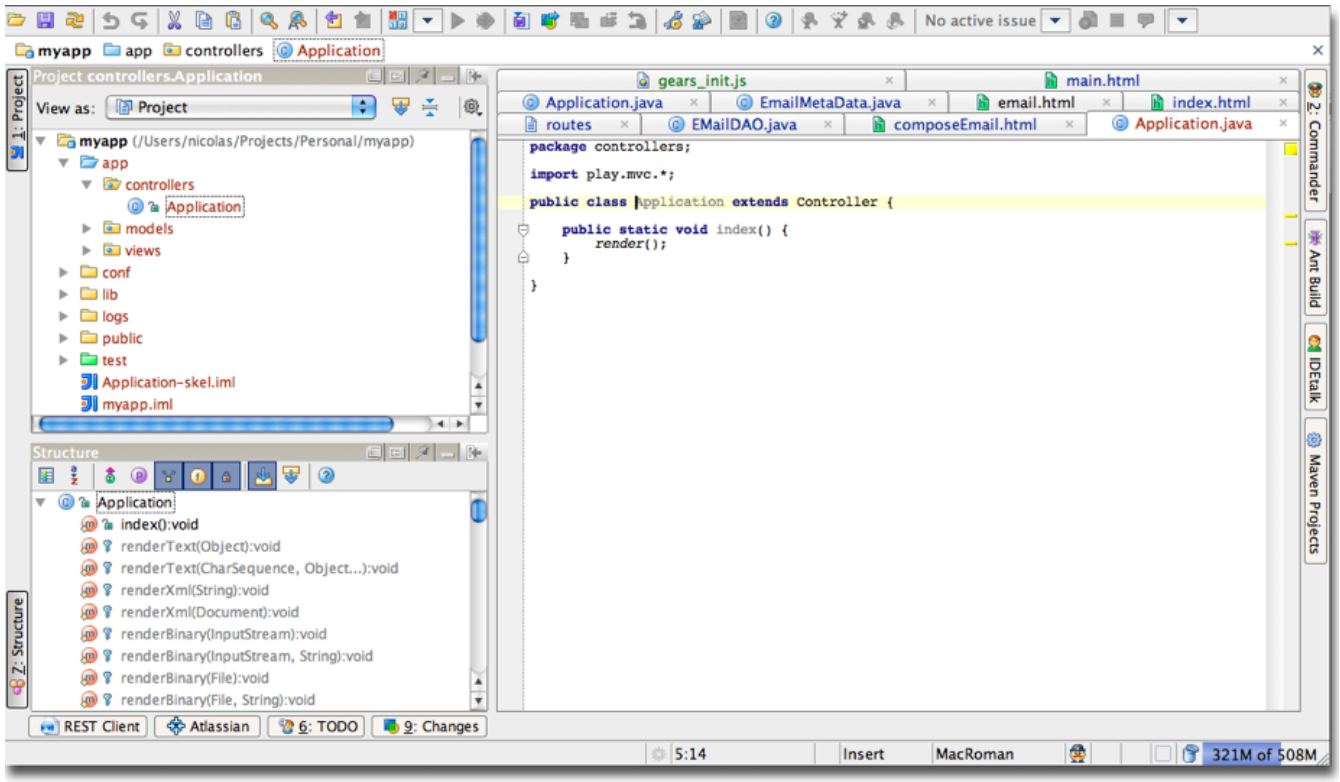
The generated configuration files contains absolute references to your framework installation. These are specific to your own installation. When you work in a team on the same application, each developer must keep his NetBeans configuration files private.

Generate configuration files for IntelliJ IDEA

Play provides a command to simplify IntelliJ IDEA configuration. To transform an existing application to a valid IntelliJ IDEA module/project, use the **idealize** command:

```
# play idealize myApp
```

Then you can just import the application in IntelliJ using the import module facility.



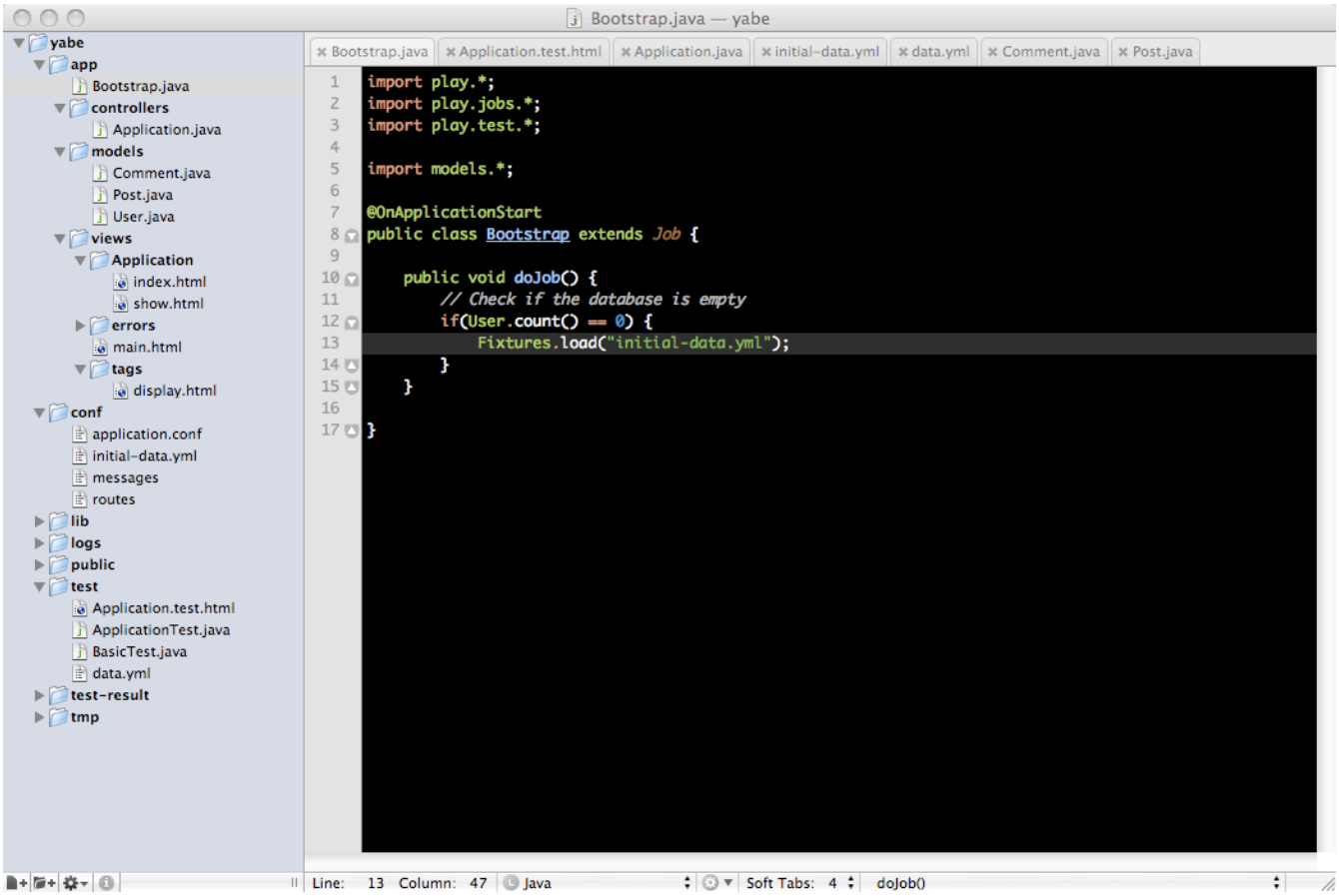
You can use the **Run** and **Debug** using the context menu.

Do not commit the .iml files when you work in a team!

The generated configuration files contains absolute references to your framework installation. These are specific to your own installation. When you work in a team on the same application, each developer must keep his IntelliJ IDEA configuration files private.

Textmate

Download and install the bundle provided for Textmate to enable syntax coloring and auto-completion. The bundle also eases navigation between controllers and views.



Manually configure your preferred editor

As Play applications are standard Java applications, you don't need a specific plug-in to work with your preferred editor. This, however, requires a little bit of knowledge of how Play works.

Classpath settings

A Play application classpath is built as follows (in this order):

- The conf/ directory for the application
- The $PLAY_PATH/framework/play.jar
- All JAR files found in your application's lib/ directory
- All JAR files found in the $PLAY_PATH/framework/lib/ directory

> **Tip**
>
> If you have any modules enabled, you will need to add all modules' libraries (from the **$module/lib/** directory) to the classpath as well.

Main class to run

To start a Play application, just run the **play.server.Server** class. Play uses the **"application.path"** system property to locate the application to run. Typically you pass this value with:

```
java -Dapplication.path="/app/path"...
```

Java agent

To enable HotSwap reloading you have to load a Java agent packaged in the **play.jar** library. Typically like this:

```
java -javaagent:"$PLAY_PATH/framework/play.jar" ...
```

It's not required, but will speed-up class reloading when it is possible.

Debugging issues

Play automatically reloads Java classes when the Java sources are modified. However as Java does not fully support class reloading, the JDPA debugger can easily get confused: breakpoint submission may fail or the debugger may stop on the wrong line when you step into the code.

To avoid this behavior, a better way is to start a fresh debugging session after a code modification. Luckily, JPDA supports the ability to connect and disconnect the debugger at any time without restarting the JVM.

So the correct workflow for debugging is:

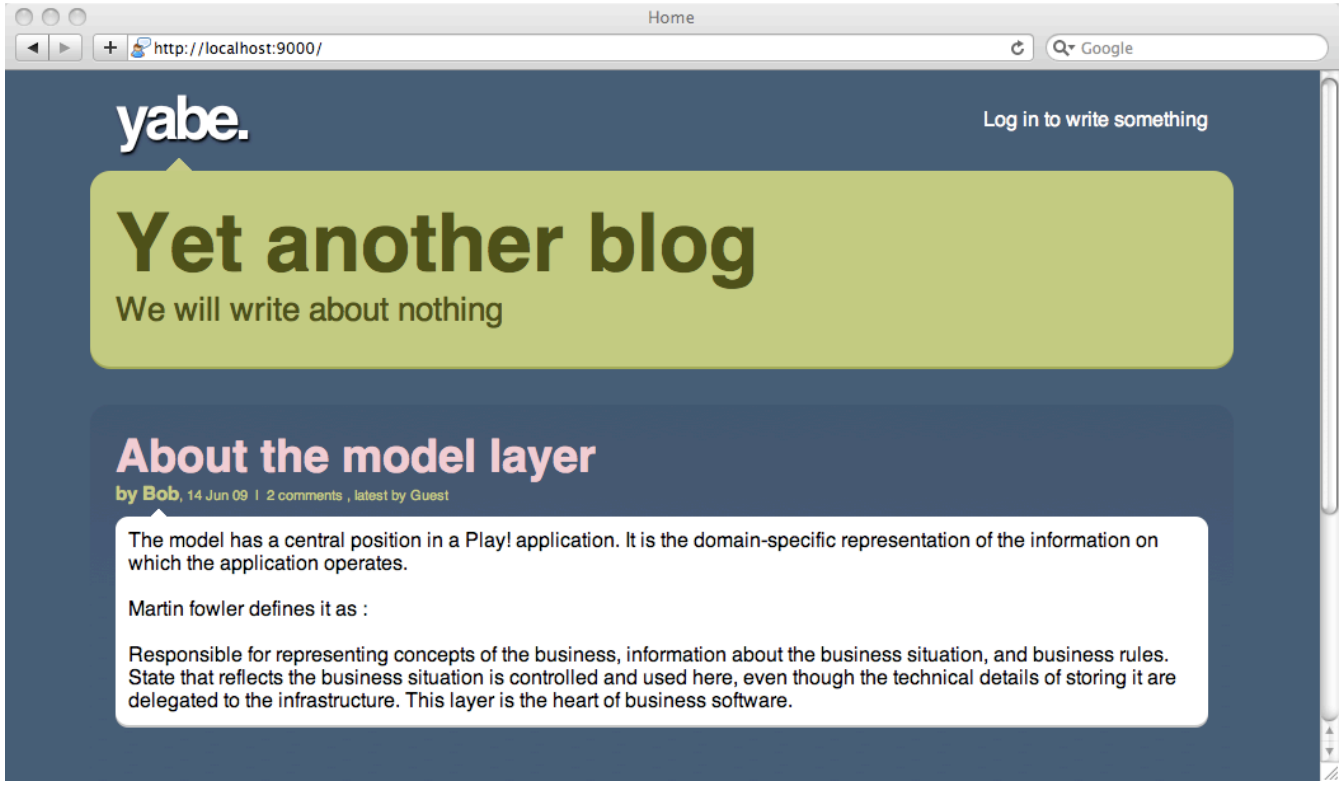
1. Make changes to your source code.
2. Refresh the browser to see the results. (At this time Play will reload your code modifications and redefine classes in the JVM.)
3. If something goes wrong and you need to debug, start a new debug session.
4. Debug and correct your code.
5. Disconnect the debugger.

By using this work-flow you will always have a debugger synchronized with the code loaded in the JVM.

Play comes with a set of sample applications in the the **samples-and-test/** directory, in the framework installation path. We use them to run our automated test suite, and you can take a look at them to discover typical Play usages.

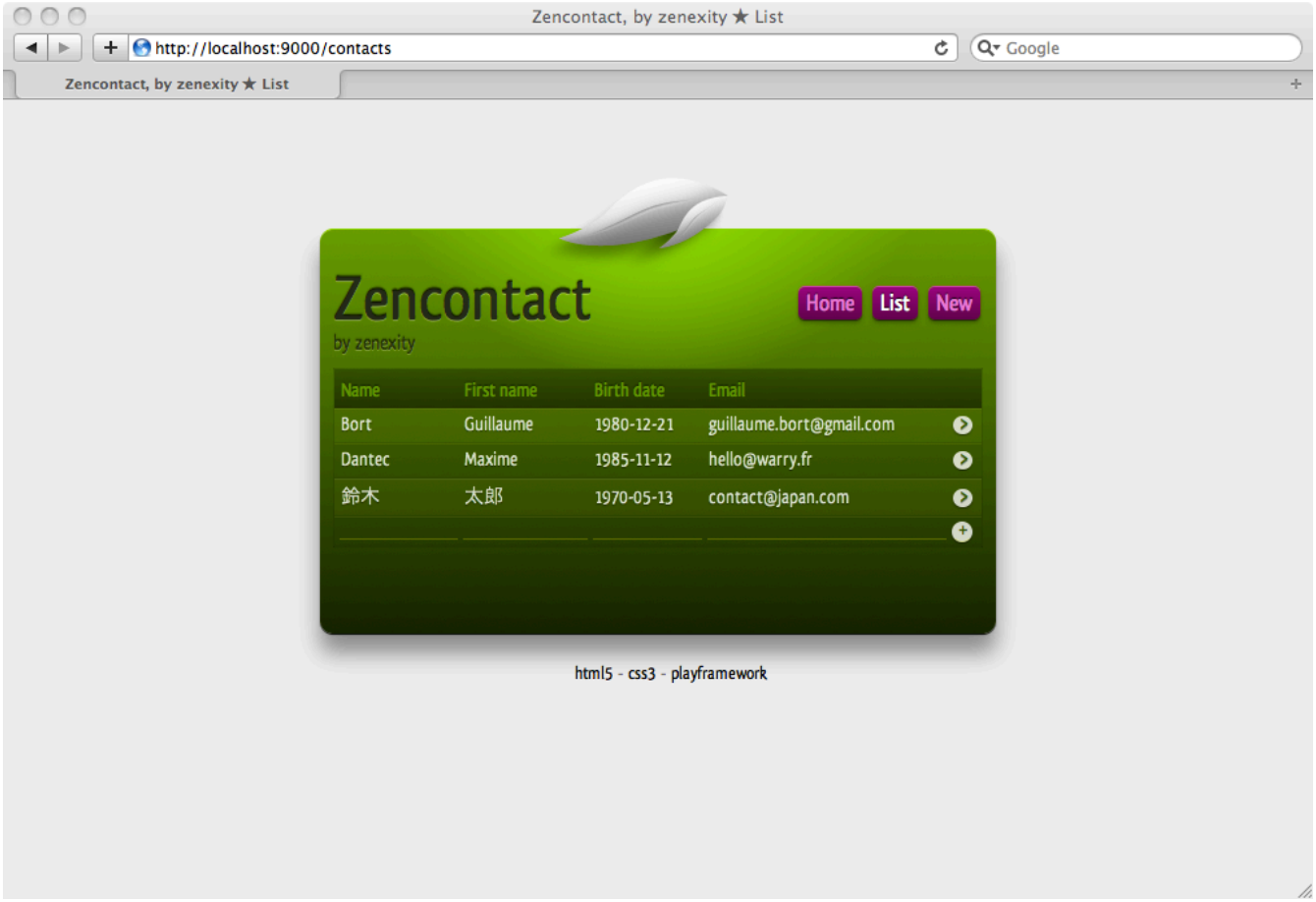
Yet Another Blog Engine

The final application from the Play guide tutorial.



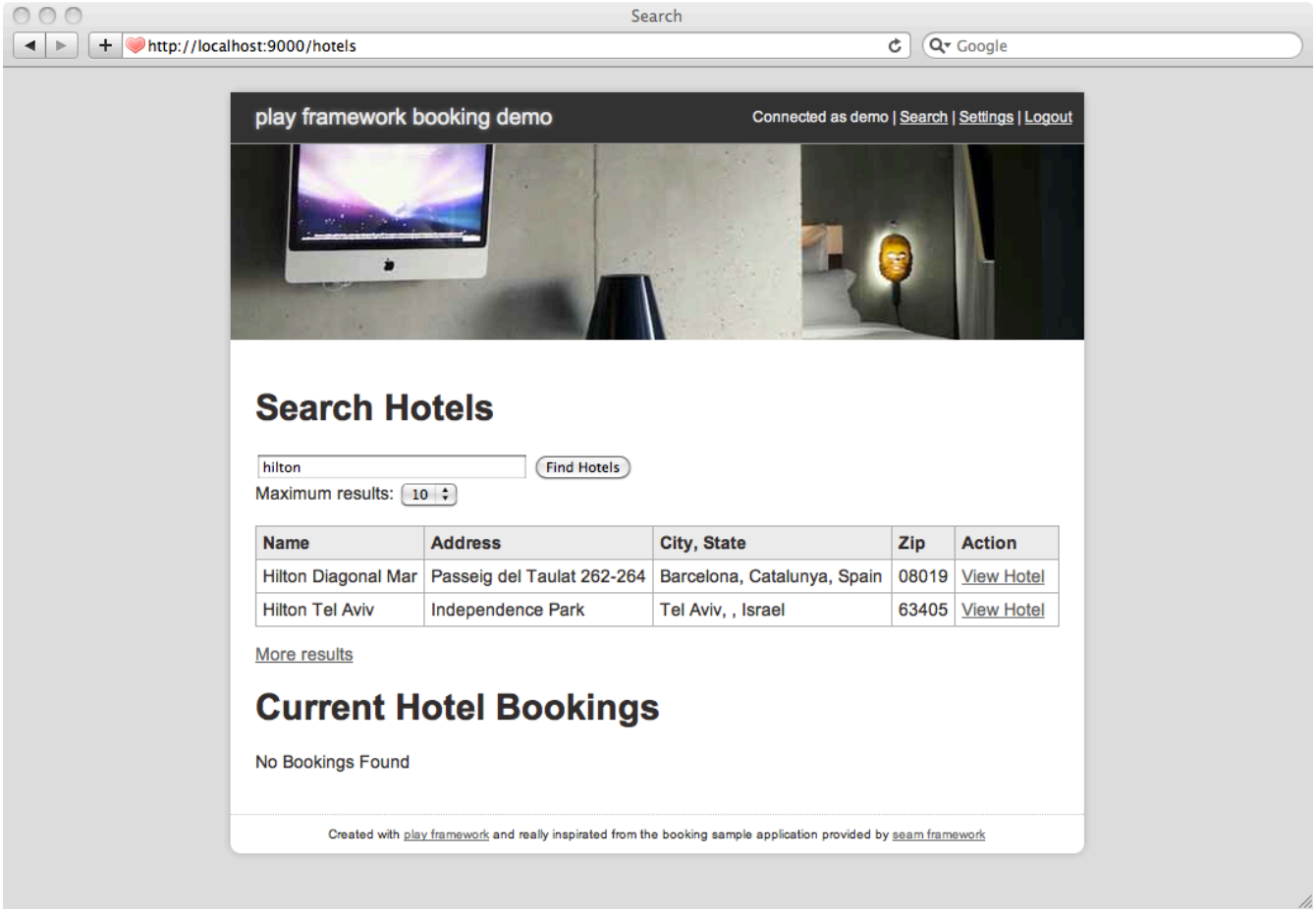
Zencontact

A simple contact management application.



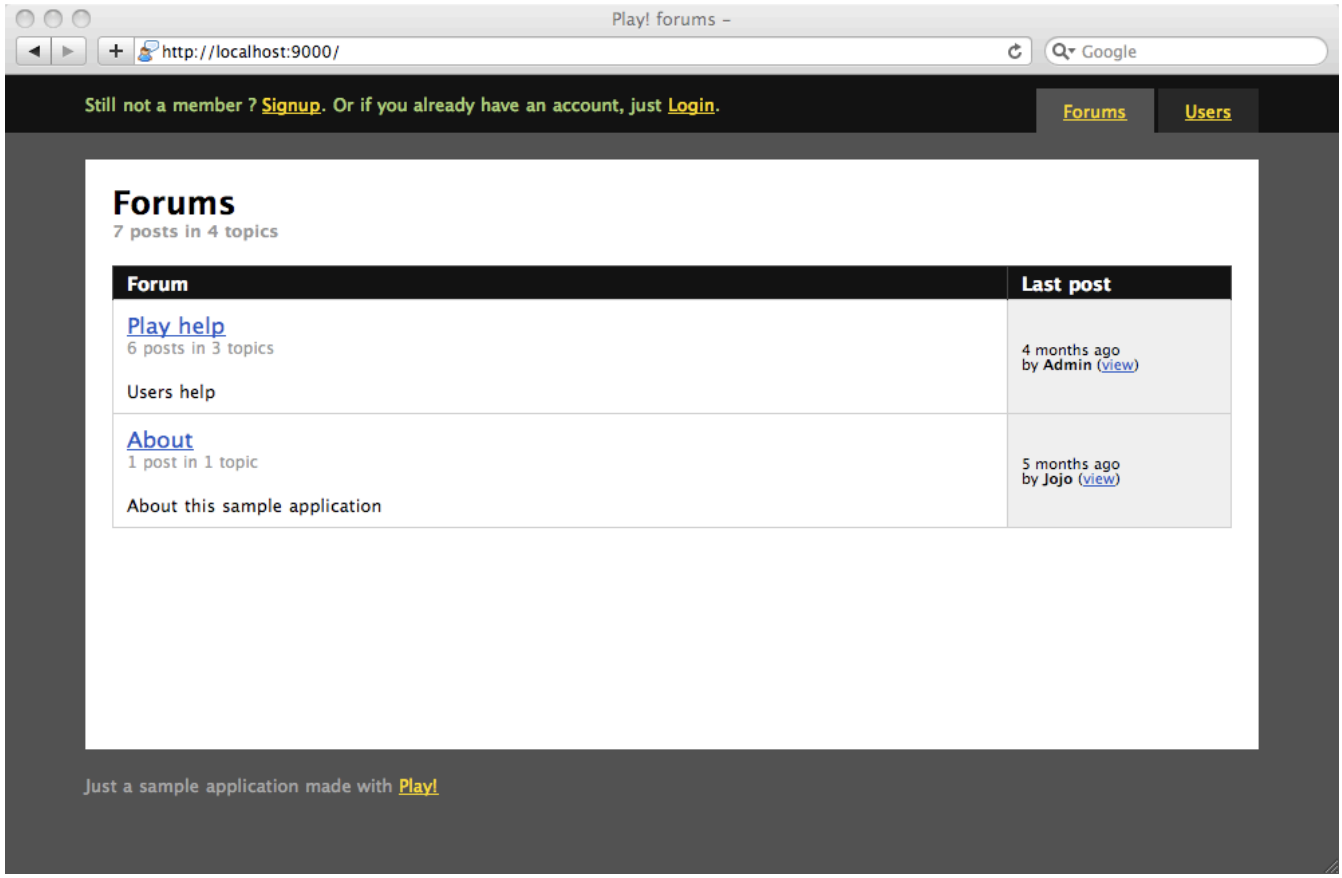
Booking

A port of the well known 'booking' application from the JBoss seam framework.



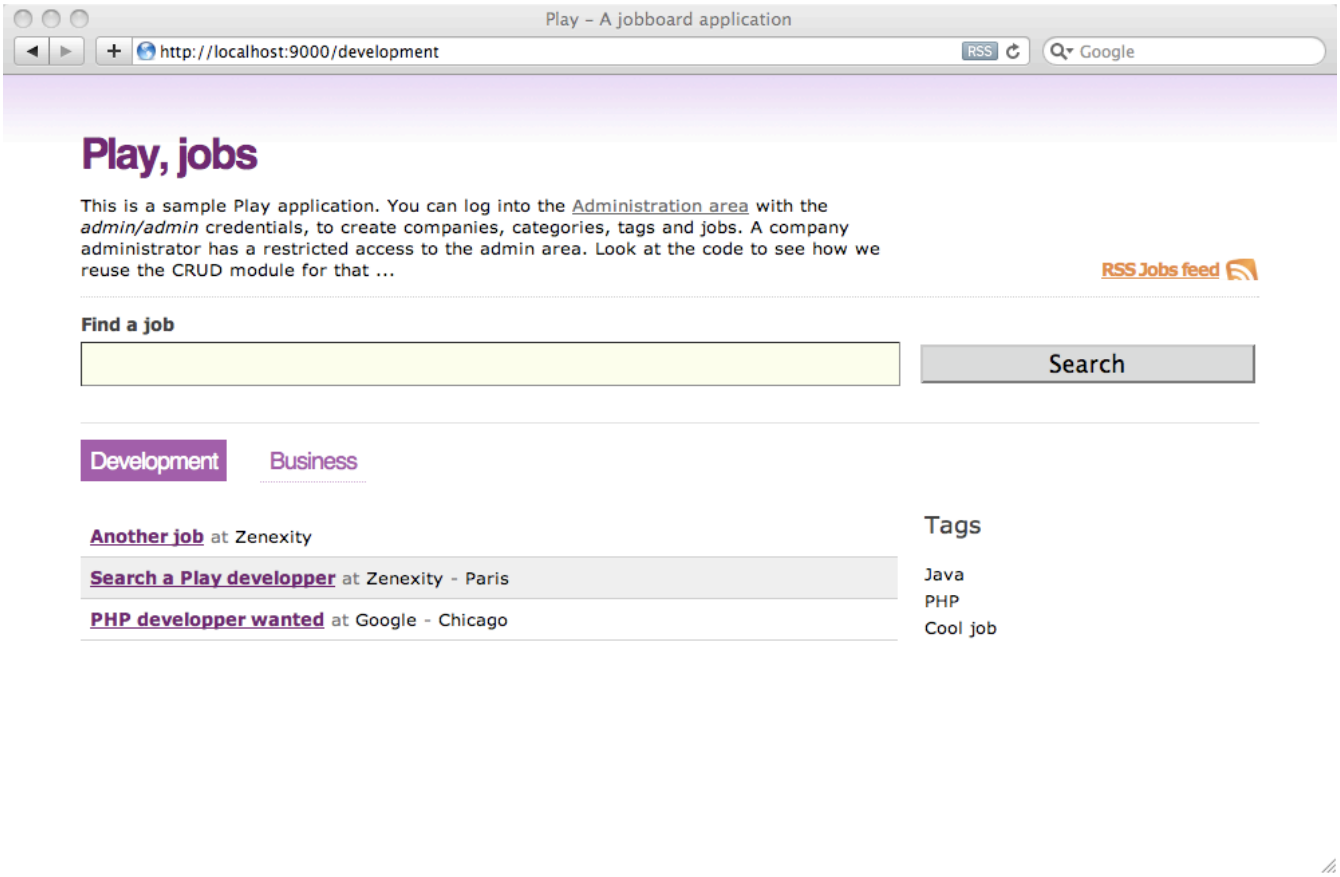
Forum

A simple forum application that uses JPA.



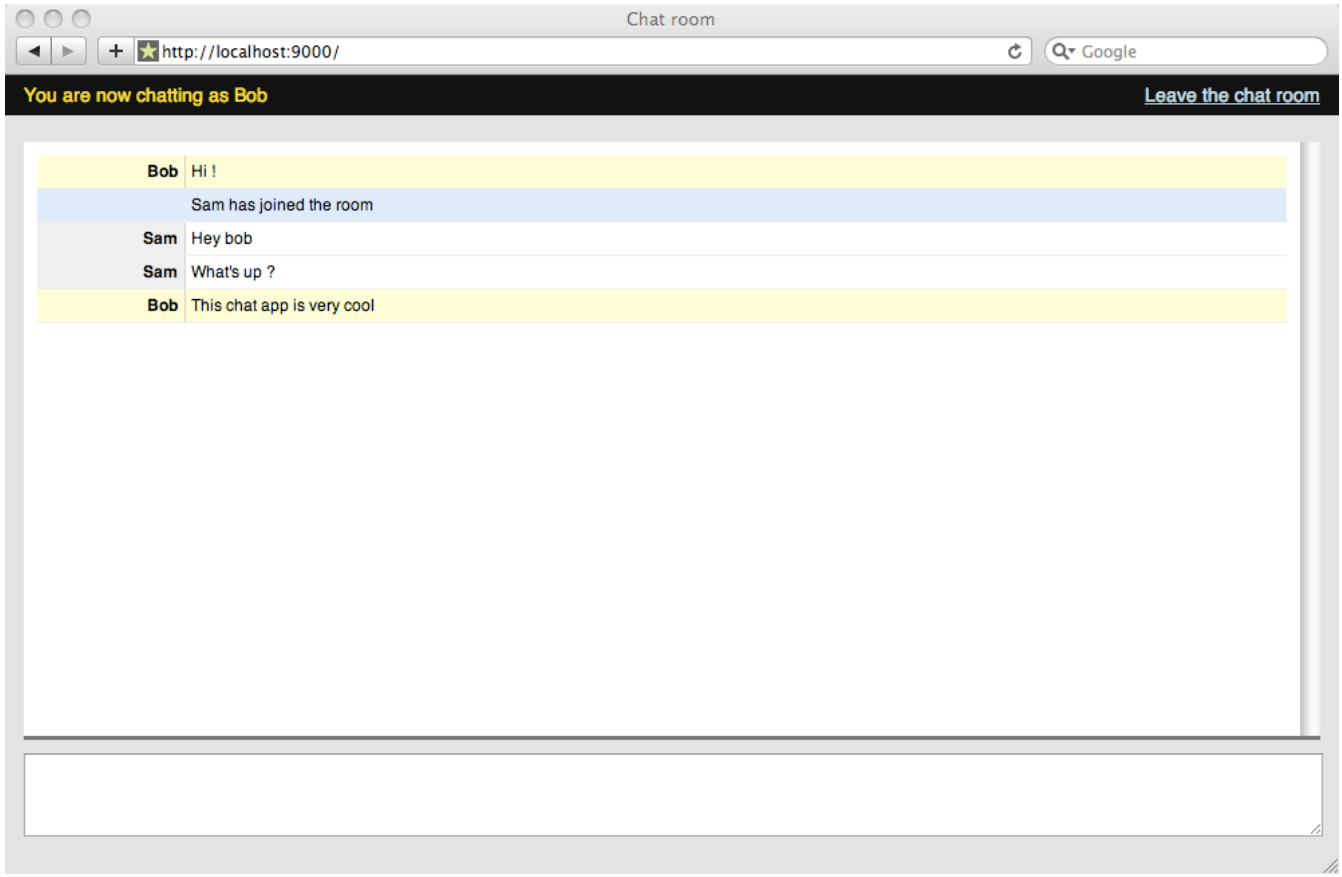
Jobboard

A fully-featured job board application that uses the CRUD module to automatically generate the administration area.



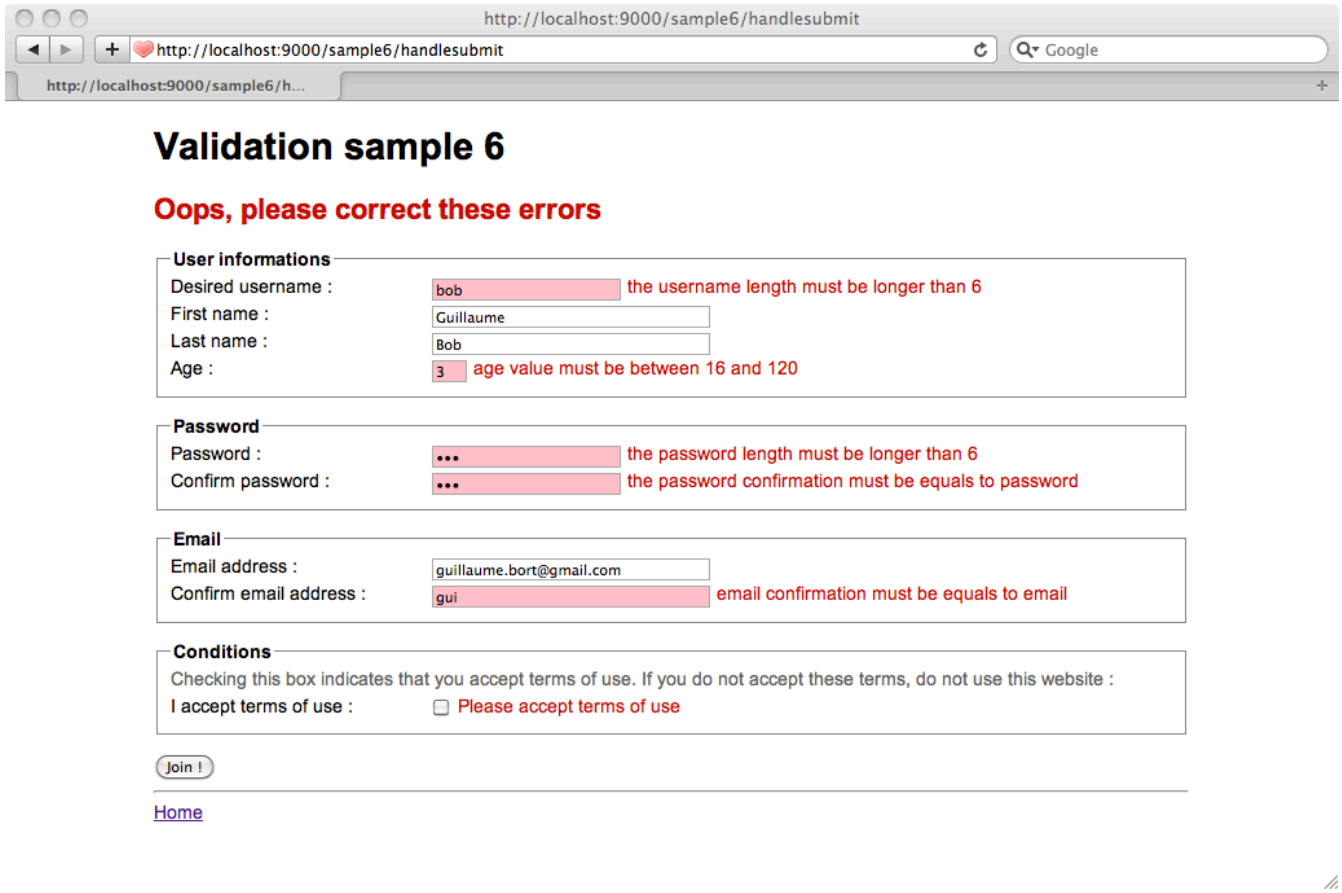
Chat

An HTTP-based chat application. It uses the asynchronous request model to be able to handle a lot of simultaneous connected users without wasting resources.



Validation

Demonstrates the different ways to create a form with validation support.


http://localhost:9000/sample6/handlesubmit
http://localhost:9000/sample6/handlesubmit
Google
http://localhost:9000/sample6/h...
Validation sample 6
Oops, please correct these errors
User informations
Desired username :
bob
the username length must be longer than 6
First name :
Guillaume
Last name :
Bob
Age :
3
age value must be between 16 and 120
Password
Password :
the password length must be longer than 6
Confirm password :
the password confirmation must be equals to password
Email
Email address :
guillaume.bort@gmail.com
Confirm email address :
gui
email confirmation must be equals to email
Conditions
Checking this box indicates that you accept terms of use. If you do not accept these terms, do not use this website :
I accept terms of use :
Please accept terms of use
Join !
Home

Introduction

In this tutorial you will learn the Play framework by coding a real web application, from start to finish. In this application, we will try to use everything you would need in a real project, while introducing good practices for Play application development.

We have split the tutorial into several independent parts. Each part will introduce more complex features, and provide everything that a real project needs: validation, error handling, a complete security framework, an automated test suite, a shiny web interface, an administration area, etc.

All the code included in this tutorial can be used for your projects. We encourage you to copy and paste snippets of code or steal whole chunks.

The project

We chose to create yet another blog engine. It's not a very imaginative choice but it will allow us to explore most of the functionality needed by a modern web application.

To make things a bit more interesting, we will manage several users with different roles (editor, admin).

We will call this blog engine project **yabe**.



This tutorial is also distributed as a sample application. You can find the final code in your Play installation's **samples-and-tests/yabe/** directory.

Prerequisites

First of all, make sure that you have a working Java installation. Play requires **Java 5 or later**.

As we will use the command line a lot, it's better to use a Unix-like OS. If you run a Windows system, it will also work fine; you'll just have to type a few commands in the command prompt.

We will assume that you already have knowledge of Java and Web development (especially HTML, CSS and JavaScript). However you don't need to have a deep knowledge of all the Java Enterprise Edition (JEE) components. Play is a 'full stack' Java framework and it provides or encapsulates all the parts of the Java API that you will need. There is no need to know how to configure a JPA entity manager or deploy a JEE component.

You will of course need a text editor. If you are accustomed to using a fully-featured Java IDE like Eclipse or NetBeans you can of course use it. However with Play you can have fun working with a simple text editor like Textmate, Emacs or VI. This is because the framework manages the compilation and the deployment process itself. As we will soon see…

Later in this tutorial we will use Lighttpd and MySql to show how to deploy a Play application in 'production' mode. But Play can work without these components so if you can't install them, it's not a problem.

Installing the Play framework

Installation is very simple. Just download the latest binary package from the download page and unzip it to any path.

> If you're using Windows, it is generally a good idea to avoid space characters in the path, so for example **c:\play** would be a better choice than **c:\Documents And Settings\user\play**.

To work efficiently, you need to add the Play directory to your working path. It allows you to just type **play** at the command prompt to use the play utility. To check that the installation worked, just open a new command line and type **play**; it should show you the play basic usage help.

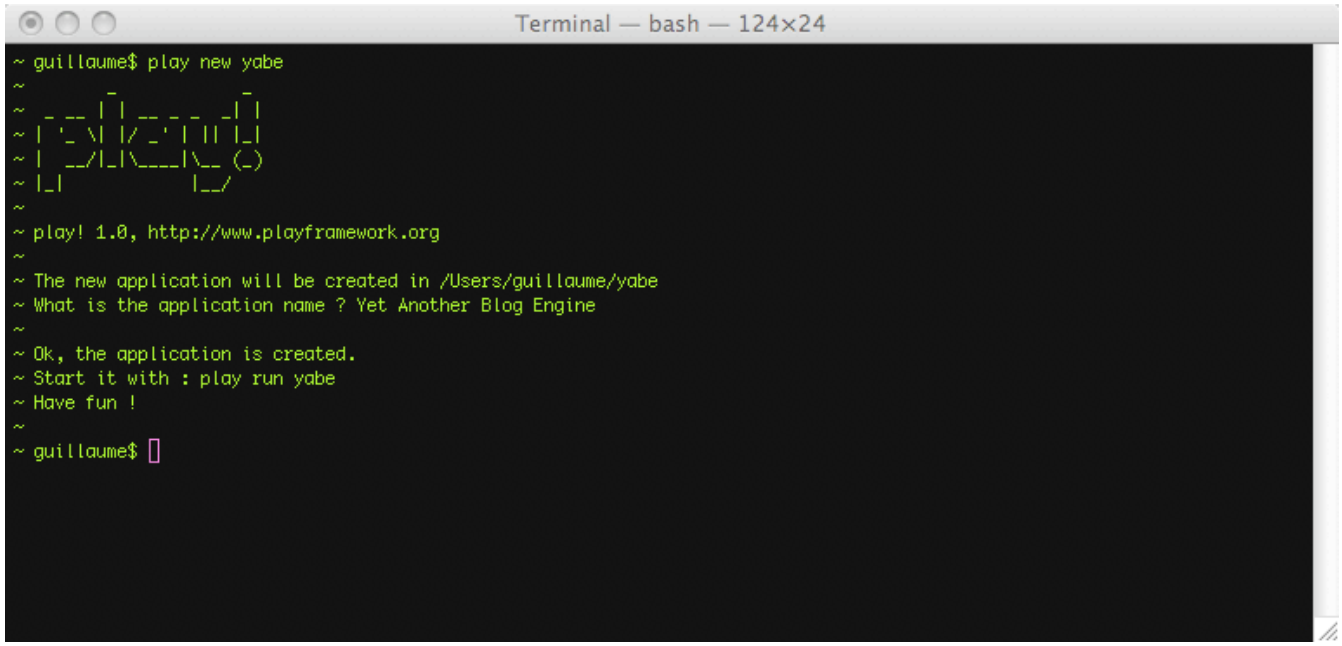
Project creation

Now that Play is correctly installed, it's time to create the blog application. Creating a Play application is pretty easy and fully managed by the play command line utility. That allows for standard project layouts between all Play applications.

Open a new command line and type:

```
~$ play new yabe
```

It will prompt you for the application full name. Type **Yet Another Blog Engine**.



The **play new** command creates a new directory **yabe/** and populates it with a series of files and directories, the most important being:

app/ contains the application's core, split between models, controllers and views directories. It can contain other Java packages as well. This is the directory where **.java** source files live.

conf/ contains all the configuration files for the application, especially the main **application.conf** file, the **routes** definition files and the **messages** files used for internationalization.

lib/ contains all optional Java libraries packaged as standard **.jar** files.

public/ contains all the publicly available resources, which includes JavaScript files, stylesheets and images directories.

test/ contains all the application tests. Tests are either written either as Java JUnit tests or as Selenium tests.

> Because **Play uses UTF-8** as single encoding, it's very important that all text files hosted in these directories are encoded using this charset. Make sure to configure your text editor accordingly.

Now if you're a seasoned Java developer, you may wonder where all the .class files go. The answer is nowhere: Play doesn't use any class files; instead it reads the Java source files directly. Under the hood we use the Eclipse compiler to compile Java sources on the fly.

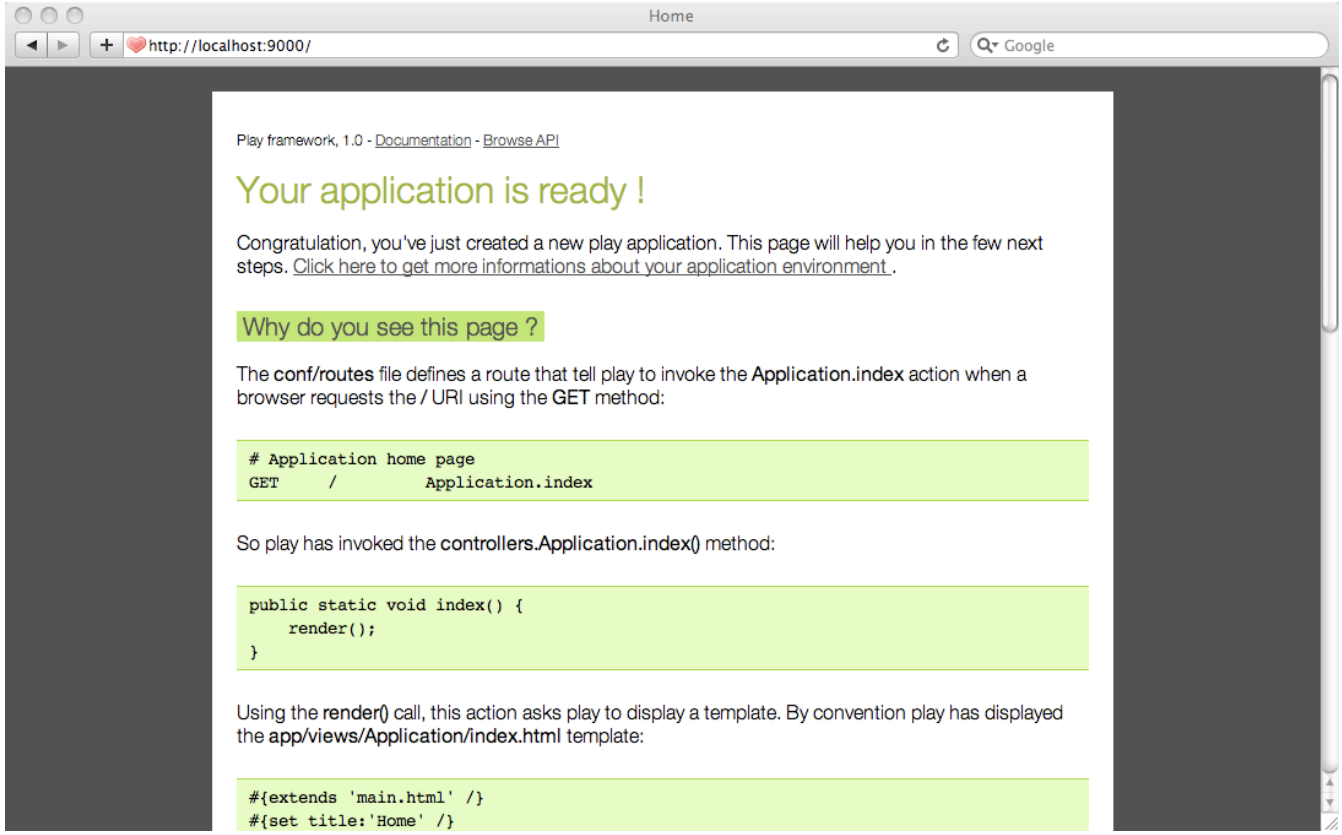
That allows two very important things in the development process. The first one is that Play will detect changes you make to any Java source file and automatically reload them at runtime. The second is that when a Java exception occurs, Play will create better error reports showing you the exact source code.

> In fact Play keeps a bytecode cache in the application's **tmp/** directory, but only to speed things up between restart on large applications. You can discard this cache using the **play clean** command if needed.

Running the application

We can now test the newly-created application. Just return to the command line, go to the newly-created **yabe/** directory and type **play run**. Play will now load the application and start a web server on port 9000.

You can see the new application by opening a browser to http://localhost:9000. A new application has a standard welcome page that just tells you that it was successfully created.



Let's see how the new application can display this page.

The main entry point of your application is the **conf/routes** file. This file defines all accessible URLs for the application. If you open the generated routes file you will see this first 'route':

```
GET           /                       Application.index
```

That simply tells Play that when the web server receives a **GET** request for the **/** path, it must call the **Application.index** Java method. In this case, **Application.index** is a shortcut for **controllers.Application.index**, because the **controllers** package is implicit.

When you create standalone Java applications you generally use a single entry point defined by a method such as:

```
public static void main(String[] args) {
  ...
}
```

A Play application has several entry points, one for each URL. We call these methods **action** methods. Action methods are defined in special classes that we call **controllers**.

Let's see what the **controllers.Application** controller looks like. Open the **yabe/app/controllers/Application.java** source file:

```
package controllers;

import play.mvc.*;

public class Application extends Controller {

        public static void index() {
                render();
        }

}
```

Notice that controller classes extend the **play.mvc.Controller** class. This class provides many useful methods for controllers, like the **render()** method we use in the index action.

The index action is defined as a **public static void** method. This is how action methods are defined. You can see that action methods are static, because the controller classes are never instantiated. They are marked public to authorize the framework to call them in response to a URL. They always return void.

The default index action is simple: it calls the **render()** method which tells Play to render a template. Using a template is the most common way (but not the only one) to generate the HTTP response.

Templates are simple text files that live in the **/app/views** directory. Because we didn't specify a template, the default one for this action will be used: **Application/index.html**

To see what the template looks like, open the **/yabe/app/views/Application/index.html** file:

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{welcome /}
```

The template content seems pretty light. In fact, all you see are Play tags. Play tags are similar to JSP tags. This is the **#{welcome /}** tag that generates the welcome message you saw in the browser.

The **#{extends /}** tag tells Play that this template inherits another template called **main.html**. Template inheritance is a powerful concept that allows you to create complex web pages by reusing common parts.

Open the **/yabe/app/views/main.html** template:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        <link rel="stylesheet" type="text/css" media="screen"
            href="@{'/public/stylesheets/main.css'}" />
        <link rel="shortcut icon" type="image/png"

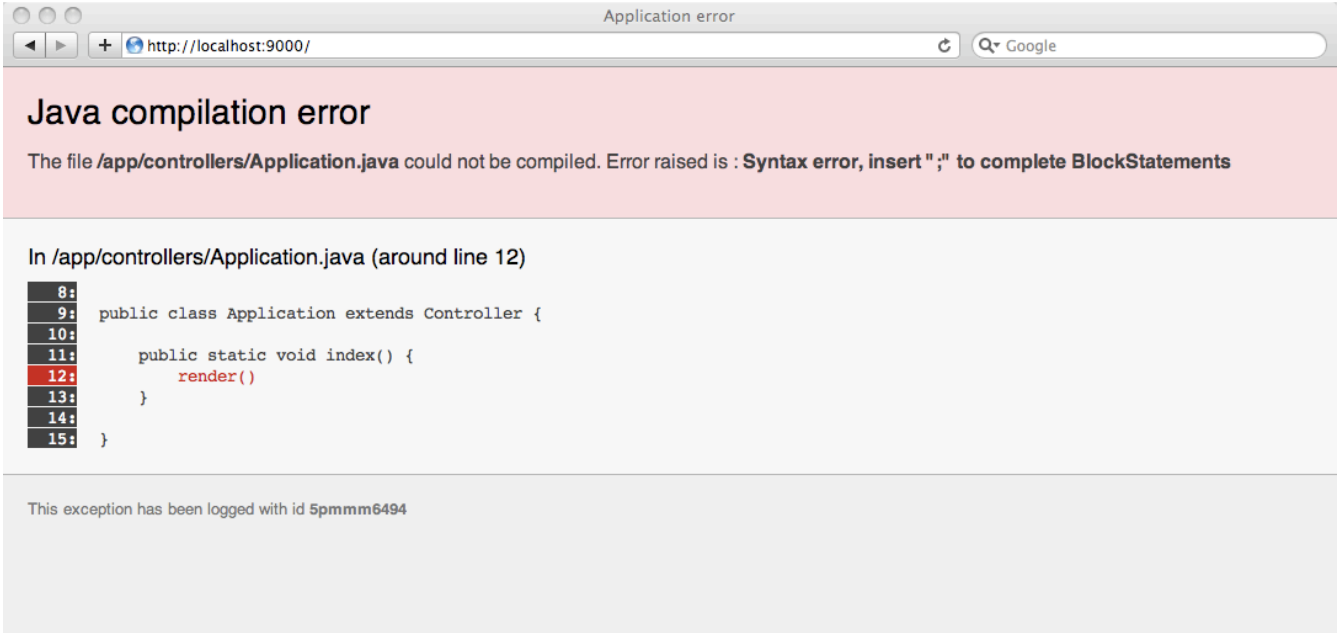
href="@{'/public/file:///Users/nicolasleroux/Projects/Personal/1.1/documentation/images/
/>
    </head>
    <body>
        #{doLayout /}
    </body>
</html>
```

Do you see the **#{doLayout /}** tag? This is where the content of **Application/index.html** will be inserted.

We can try to edit the controller file to see how Play automatically reloads it. Open the **yabe/app/controllers/Application.java** file in a text editor, and add a mistake by removing the trailing semicolon after the **render()** call:

```
public static void index() {
    render()
}
```

Go to the browser and refresh the page. You can see that Play detected the change and tried to reload the Application controller. But because you made a mistake, you get a compilation error.



Ok, let's correct the error, and make a real modification:

```
public static void index() {
    System.out.println("Yop");
    render();
}
```

This time, Play has correctly reloaded the controller and replaced the old code in the JVM. Each request to the **/** URL will output the 'Yop' message to the console.

You can remove this useless line, and now edit the **yabe/app/views/Application/index.html** template to replace the welcome message:

```
#{extends 'main.html' /}
#{set title:'Home' /}

<h1>A blog will be there</h1>
```

Like for Java code changes, just refresh the page in the browser to see the modification.

> We will now start to code the blog application. You can either continue to work with a text editor or open the project in a Java IDE like Eclipse or NetBeans. If you want to set-up a Java IDE, please check this page.

Setting-up the database

One more thing before starting to code. For the blog engine, we will need a database. For development purposes, Play comes with a stand alone SQL database management system called HSQLDB. This is the best way to start a project before switching to a more robust database if needed. You can choose to have either an in-memory database or a filesystem database that will keep your data between application restarts.

At the beginning, we will do a lot of testing and changes in the application model. For that reason, it's better to use an in-memory database so we always start with a fresh data set.

To set-up the database, open the **yabe/conf/application.conf** file and uncomment this line:

```
db=mem
```

As you can see in the comments, you can easily set-up any JDBC compliant database and even configure the connection pool.

Now, go back to your browser and refresh the welcome page. Play will automatically start the database. Check for this line in the application logs:

```
INFO  ~ Connected to jdbc:hsqldb:mem:playembed
```

Using a version control system to track changes

When you work on a project, it's highly recommended to store your source code in a version control system (VCS). It allows you to revert to a previous version if a change breaks something, work with several people and give access to all the successive versions of the application. Of course you can use any VCS to store your project, but here we will use Bazaar as an example. Bazaar is a distributed source version control system.

Installing Bazaar is beyond the scope of this tutorial but it is very easy on any system. Once you have a working installation of Bazaar, go to the blog directory and init the application versioning by typing:

```
$ bzr init
```

When storing a Play application in a VCS, it's important to exclude the **tmp/** and **logs/** directories.

```
$ bzr ignore tmp
$ bzr ignore logs
```

Now we can commit our first blog engine version:

```
$ bzr add
$ bzr commit -m "YABE inital version"
```

Version 1 is committed and we now have a solid foundation for our project.

Go to the next part.

Here we will start to write the model for our blog engine.

Introduction to JPA

The model layer has a central position in a Play application (and in fact in all well-designed applications). It is the domain-specific representation of the information on which the application operates. As we want to create a blog engine, the model layer will certainly contain classes like User, Post and Comment.

Because most model objects need to survive between application restarts, we have to save them in a persistent datastore. A common choice is to use a relational database. But because Java is an object oriented language, we will use an object-relational mapper to help reduce the impedance mismatch.

The Java Persistence API (JPA) is a Java specification that defines a standard API for object-relational mapping. As an implementation of JPA, Play uses the well-known Hibernate framework. One advantage of using JPA over the standard Hibernate API is that all the 'mapping' is declared directly in the Java objects.

If you have used Hibernate or JPA before you will be surprised by the simplicity added by Play. No need to configure anything; JPA just works out of the box with Play.

If you don't know JPA, you can read some of these simple presentations before continuing.

The User class

We will start to code the blog engine by creating the User class. Create a new file **/yabe/app/models/User.java**, and declare a first implementation of the User class:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;

@Entity
public class User extends Model {

    public String email;
    public String password;
    public String fullname;
    public boolean isAdmin;

    public User(String email, String password, String fullname) {
        this.email = email;
        this.password = password;
        this.fullname = fullname;
    }

}
```

The **@Entity** annotation marks this class as a managed JPA entity, and the **Model** superclass automatically provides a set of useful JPA helpers that we will discover later. All fields of this class will be automatically persisted to the database.

> It's not required that your model objects extend the **play.db.jpa.Model** class. You can use plain JPA as well. But extending this class is a good choice in most cases as it will make a lot of the JPA stuff easier.

If you have used JPA before, you know that every JPA entity must provide an **@Id** property. Here the Model superclass provides an automatically generated numeric ID, and in most cases this is good enough.

> Don't think about this provided **id** field as a **functional identifier** but as a **technical identifier**. It is generally a good idea to keep both concepts separated and to keep an automatically generated numeric ID as a technical identifier.

Now if you're a Java developer with any experience at all, warning bells are probably clanging like mad at the sight of a public variable. In Java (as in other object-oriented languages), best practice says to make all fields private and provide accessors and mutators. This is to promote encapsulation, a concept critical to object oriented design. In fact, Play takes care of that for you and automatically generates getters and setters while preserving encapsulation; we will see how it works later in this tutorial.

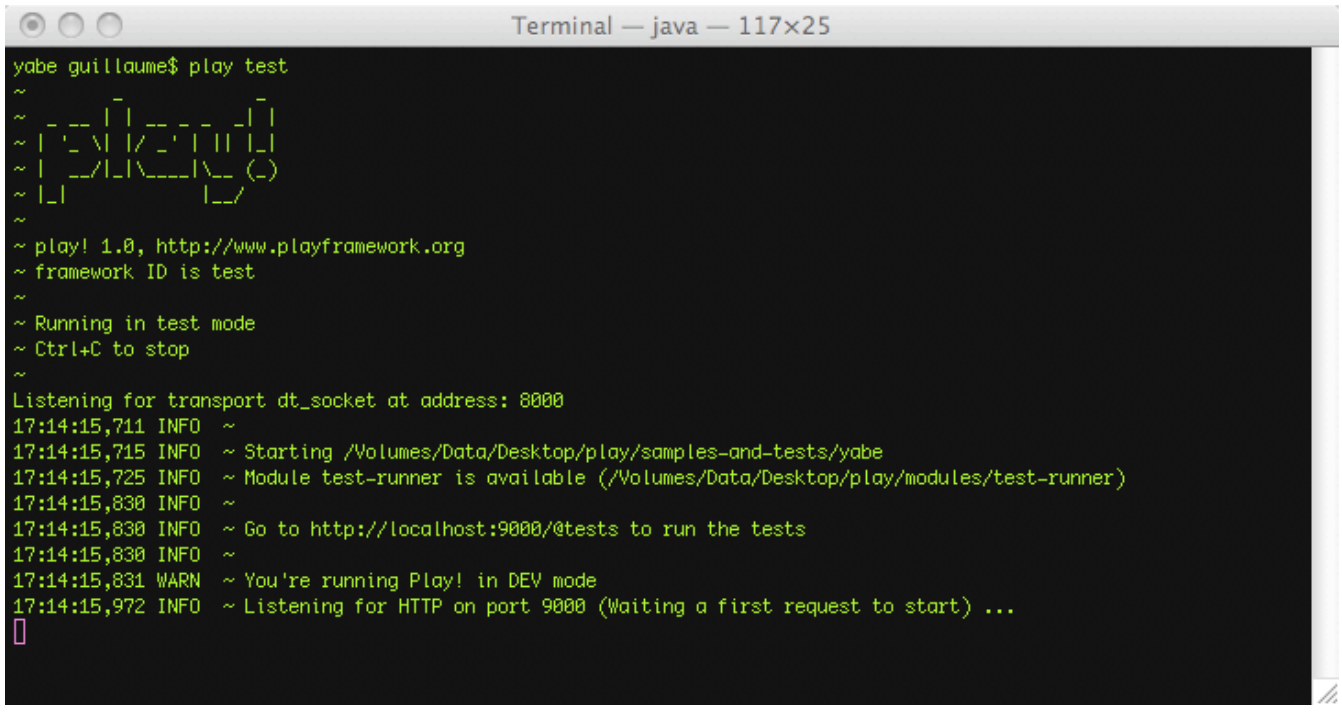
You can now refresh the application home page, to see the result. In fact, unless you made a mistake, you should not see any change: Play has automatically compiled and loaded the User class, but this does not add any new features to the application.

Writing the first test

A good way to test the newly created User class is to write a JUnit test case. It will allow you to incrementally complete the application model and ensure that all is fine.

To run a test case, you need to start the application in a special 'test' mode. Stop the currently running application, open a command line and type:

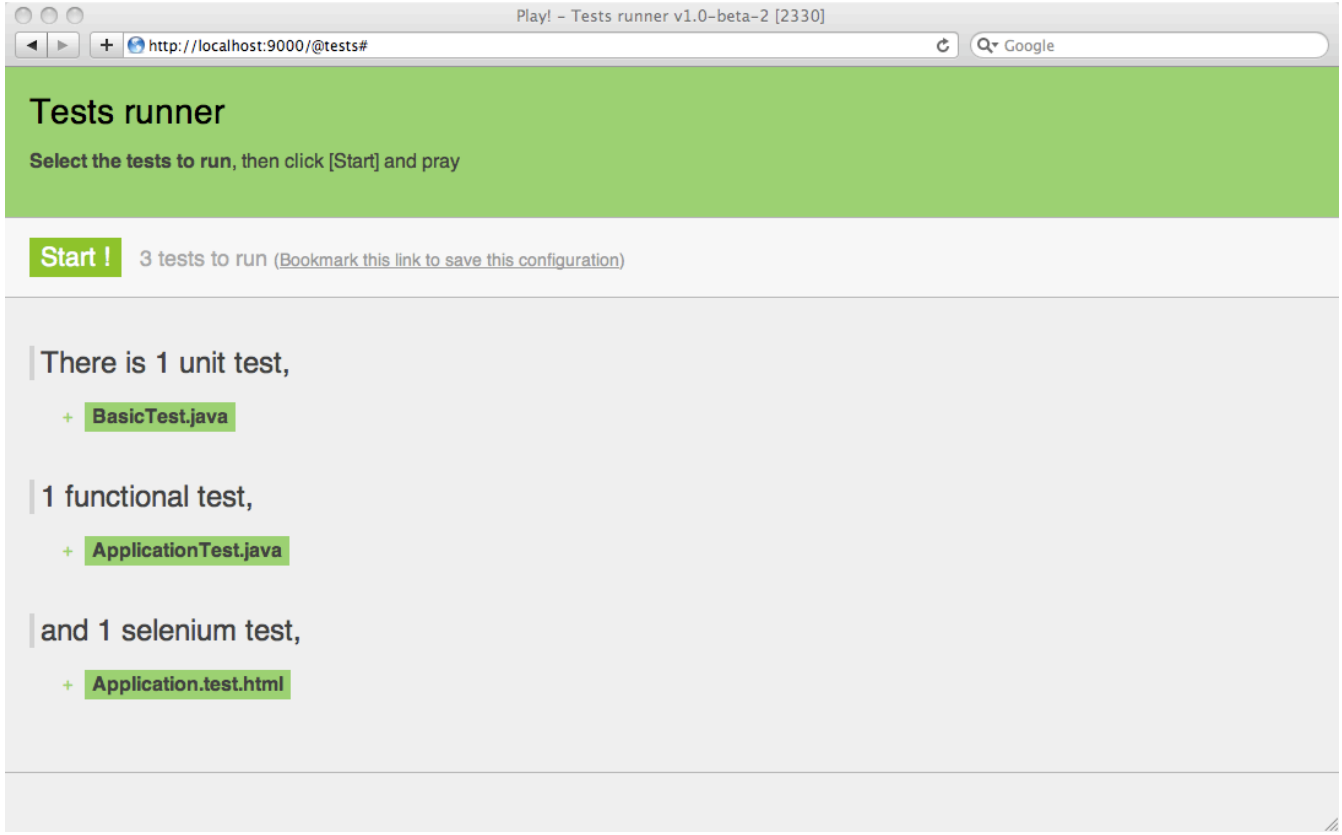
```
~$ play test
```



The **play test** command is almost the same as **play run**, except that it loads a test runner module that allows you to run test suite directly from a browser.

> When you run a Play application in **test mode**, Play will automatically switch to the **test** framework ID and load the **application.conf** file accordingly. Check the framework ID documentation for more information.

Open a browser to the http://localhost:9000/@tests URL to see the test runner. Try to select all the default tests and run them; all should be green… But these default tests don't really test anything.



To test the model part of the application we will use a JUnit test. As you can see, a default **BasicTests.java** already exists, so let's open it (**/yabe/test/BasicTest.java**):

```
import org.junit.*;
import play.test.*;
import models.*;

public class BasicTest extends UnitTest {

    @Test
    public void aVeryImportantThingToTest() {
        assertEquals(2, 1 + 1);
    }

}
```

Remove the useless default test (**aVeryImportantThingToTest**) and create a test that tries to create a new user and retrieve it:

```
@Test
public void createAndRetrieveUser() {
    // Create a new user and save it
    new User("bob@gmail.com", "secret", "Bob").save();

    // Retrieve the user with e-mail address bob@gmail.com
    User bob = User.find("byEmail", "bob@gmail.com").first();

    // Test
    assertNotNull(bob);
    assertEquals("Bob", bob.fullname);
}
```

As you can see, the Model superclass gives us two very useful methods: **save()** and **find()**.

> You can read more about the Model class' methods in the Play manual's JPA support chapter.

Select the **BasicTests.java** in the test runner, click start and check that all is green.

We will need a method on the User class that checks if a user with a specified username and password exists. Let's write it and test it.

In the **User.java** source, add the **connect()** method:

```
public static User connect(String email, String password) {
    return find("byEmailAndPassword", email, password).first();
}
```

And now the test case:

```
@Test
public void tryConnectAsUser() {
    // Create a new user and save it
    new User("bob@gmail.com", "secret", "Bob").save();

    // Test
    assertNotNull(User.connect("bob@gmail.com", "secret"));
    assertNull(User.connect("bob@gmail.com", "badpassword"));
    assertNull(User.connect("tom@gmail.com", "secret"));
}
```

Each time you make a modification you can run all the tests from the Play test runner to make sure you didn't break anything.

The Post class

The **Post** class will represent blog posts. Let's write a first implementation:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;

@Entity
public class Post extends Model {

    public String title;
    public Date postedAt;

    @Lob
    public String content;

    @ManyToOne
    public User author;

    public Post(User author, String title, String content) {
        this.author = author;
        this.title = title;
        this.content = content;
        this.postedAt = new Date();
    }

}
```

Here we use the **@Lob** annotation to tell JPA to use a large text database type to store the post content. We have declared the relation to the **User** class using **@ManyToOne**. That means that each **Post** is authored by a single **User**, and that each **User** can author several **Posts**.

We will write a new test case to check that the **Post** class works as expected. But before writing more tests, we need to do something in the JUnit test class. In the current test, the database content is never deleted, so each new run creates more and more objects. This will soon become problematic, when more advanced tests count objects to check that all is fine.

So let's write a JUnit **setup()** method to delete the database before each test:

```
public class BasicTest extends UnitTest {

    @Before
    public void setup() {
        Fixtures.deleteAll();
    }

    ...

}
```

The **@Before** concept is a core concept of the JUnit testing tool.

As you can see, the **Fixtures** class is a helper to deal with your database during tests. Run the test again to check that you haven't broken anything, and start to write the next test:

```
@Test
public void createPost() {
    // Create a new user and save it
    User bob = new User("bob@gmail.com", "secret", "Bob").save();

    // Create a new post
    new Post(bob, "My first post", "Hello world").save();

    // Test that the post has been created
    assertEquals(1, Post.count());

    // Retrieve all posts created by Bob
    List<Post> bobPosts = Post.find("byAuthor", bob).fetch();

    // Tests
    assertEquals(1, bobPosts.size());
    Post firstPost = bobPosts.get(0);
    assertNotNull(firstPost);
    assertEquals(bob, firstPost.author);
    assertEquals("My first post", firstPost.title);
    assertEquals("Hello world", firstPost.content);
    assertNotNull(firstPost.postedAt);
}
```

Don't forget to import the **java.util.List** or will get a compilation error.

Adding Comments

The last thing that we need to add to this first model draft is the ability to attach comments to posts.

Creating the **Comment** class is pretty straightforward.

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;

@Entity
public class Comment extends Model {

    public String author;
    public Date postedAt;

    @Lob
    public String content;

    @ManyToOne
    public Post post;

    public Comment(Post post, String author, String content) {
        this.post = post;
        this.author = author;
        this.content = content;
        this.postedAt = new Date();
    }

}
```

Let's write a first test case:

```
@Test
public void postComments() {
    // Create a new user and save it
    User bob = new User("bob@gmail.com", "secret", "Bob").save();

    // Create a new post
    Post bobPost = new Post(bob, "My first post", "Hello world").save();

    // Post a first comment
    new Comment(bobPost, "Jeff", "Nice post").save();
    new Comment(bobPost, "Tom", "I knew that !").save();

    // Retrieve all comments
    List<Comment> bobPostComments = Comment.find("byPost", bobPost).fetch();

    // Tests
    assertEquals(2, bobPostComments.size());

    Comment firstComment = bobPostComments.get(0);
    assertNotNull(firstComment);
    assertEquals("Jeff", firstComment.author);
    assertEquals("Nice post", firstComment.content);
    assertNotNull(firstComment.postedAt);

    Comment secondComment = bobPostComments.get(1);
    assertNotNull(secondComment);
    assertEquals("Tom", secondComment.author);
    assertEquals("I knew that !", secondComment.content);
    assertNotNull(secondComment.postedAt);
}
```

You can see that navigation between **Post** and **Comments** is not very easy: we need to use a query to retrieve all comments attached to a **Post**. We can do better by setting up the other side of the relationship with the **Post** class.

Add the **comments** field to the **Post** class:

```
...
@OneToMany(mappedBy="post", cascade=CascadeType.ALL)
public List<Comment> comments;

public Post(User author, String title, String content) {
    this.comments = new ArrayList<Comment>();
    this.author = author;
    this.title = title;
    this.content = content;
    this.postedAt = new Date();
}
...
```

Note how we have used the **mappedBy** attribute to tell JPA that the **Comment** class's post field maintains the relationship. When you define a bi-directional relation with JPA it is very important to tell it which side will maintain the relationship. In this case, since the **Comments** belong to the **Post**, it's better that the **Comment** class maintains the relationship.

We have set the **cascade** property to tell JPA that we want **Post** deletion be cascaded to **comments**. This way, if you delete a post, all related comments will be deleted as well.

With this new relationship, we will add a helper method to the **Post** class to simplify adding comments:

```
public Post addComment(String author, String content) {
    Comment newComment = new Comment(this, author, content).save();
    this.comments.add(newComment);
    this.save();
    return this;
}
```

Let's write another test case to check that it works:

```
@Test
public void useTheCommentsRelation() {
    // Create a new user and save it
    User bob = new User("bob@gmail.com", "secret", "Bob").save();

    // Create a new post
    Post bobPost = new Post(bob, "My first post", "Hello world").save();

    // Post a first comment
    bobPost.addComment("Jeff", "Nice post");
    bobPost.addComment("Tom", "I knew that !");

    // Count things
    assertEquals(1, User.count());
    assertEquals(1, Post.count());
    assertEquals(2, Comment.count());

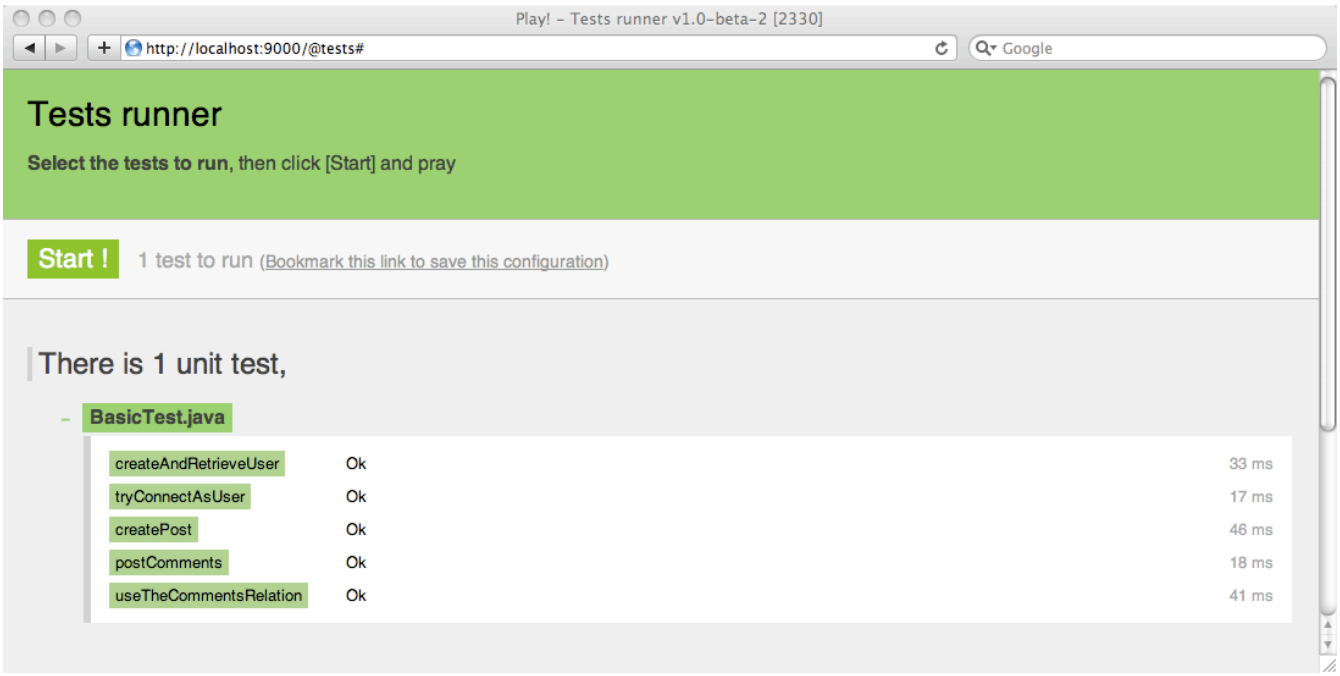
    // Retrieve Bob's post
    bobPost = Post.find("byAuthor", bob).first();
    assertNotNull(bobPost);

    // Navigate to comments
    assertEquals(2, bobPost.comments.size());
    assertEquals("Jeff", bobPost.comments.get(0).author);

    // Delete the post
    bobPost.delete();

    // Check that all comments have been deleted
    assertEquals(1, User.count());
    assertEquals(0, Post.count());
    assertEquals(0, Comment.count());
}
```

Is it green?



Using Fixtures to write more complicated tests

When you start to write more complex tests, you often need a set of data to test on. Fixtures let you describe your model in a YAML file and load it at any time before a test.

Edit the **/yabe/test/data.yml** file and start to describe a User:

```
User(bob):
    email: bob@gmail.com
    password: secret
    fullname: Bob

...
```

Well, because the **data.yml** file is a litle big, you can download it here.

Now we create create a test case that loads this data and runs some assertions over it:

```
@Test
public void fullTest() {
    Fixtures.load("data.yml");

    // Count things
    assertEquals(2, User.count());
    assertEquals(3, Post.count());
    assertEquals(3, Comment.count());

    // Try to connect as users
    assertNotNull(User.connect("bob@gmail.com", "secret"));
    assertNotNull(User.connect("jeff@gmail.com", "secret"));
    assertNull(User.connect("jeff@gmail.com", "badpassword"));
    assertNull(User.connect("tom@gmail.com", "secret"));

    // Find all of Bob's posts
    List<Post> bobPosts = Post.find("author.email", "bob@gmail.com").fetch();
    assertEquals(2, bobPosts.size());

    // Find all comments related to Bob's posts
    List<Comment> bobComments = Comment.find("post.author.email",
"bob@gmail.com").fetch();
    assertEquals(3, bobComments.size());

    // Find the most recent post
    Post frontPost = Post.find("order by postedAt desc").first();
    assertNotNull(frontPost);
    assertEquals("About the model layer", frontPost.title);

    // Check that this post has two comments
    assertEquals(2, frontPost.comments.size());

    // Post a new comment
    frontPost.addComment("Jim", "Hello guys");
    assertEquals(3, frontPost.comments.size());
    assertEquals(4, Comment.count());
}
```

Save your work

We have now finished a huge part of the blog engine. Now that we have created and tested all these things, we can start to develop the web application itself.

But before continuing, it's time to save your work using Bazaar. Open a command line and type **bzr st** to see the modifications made since the last commit:

```
$ bzr st
```

As you can see, some new files are not under version control. The **test-result** folder doesn't need to be versioned, so let's ignore it.

```
$ bzr ignore test-result
```

Add all the other files to version control using **bzr add**.

```
$ bzr add
```

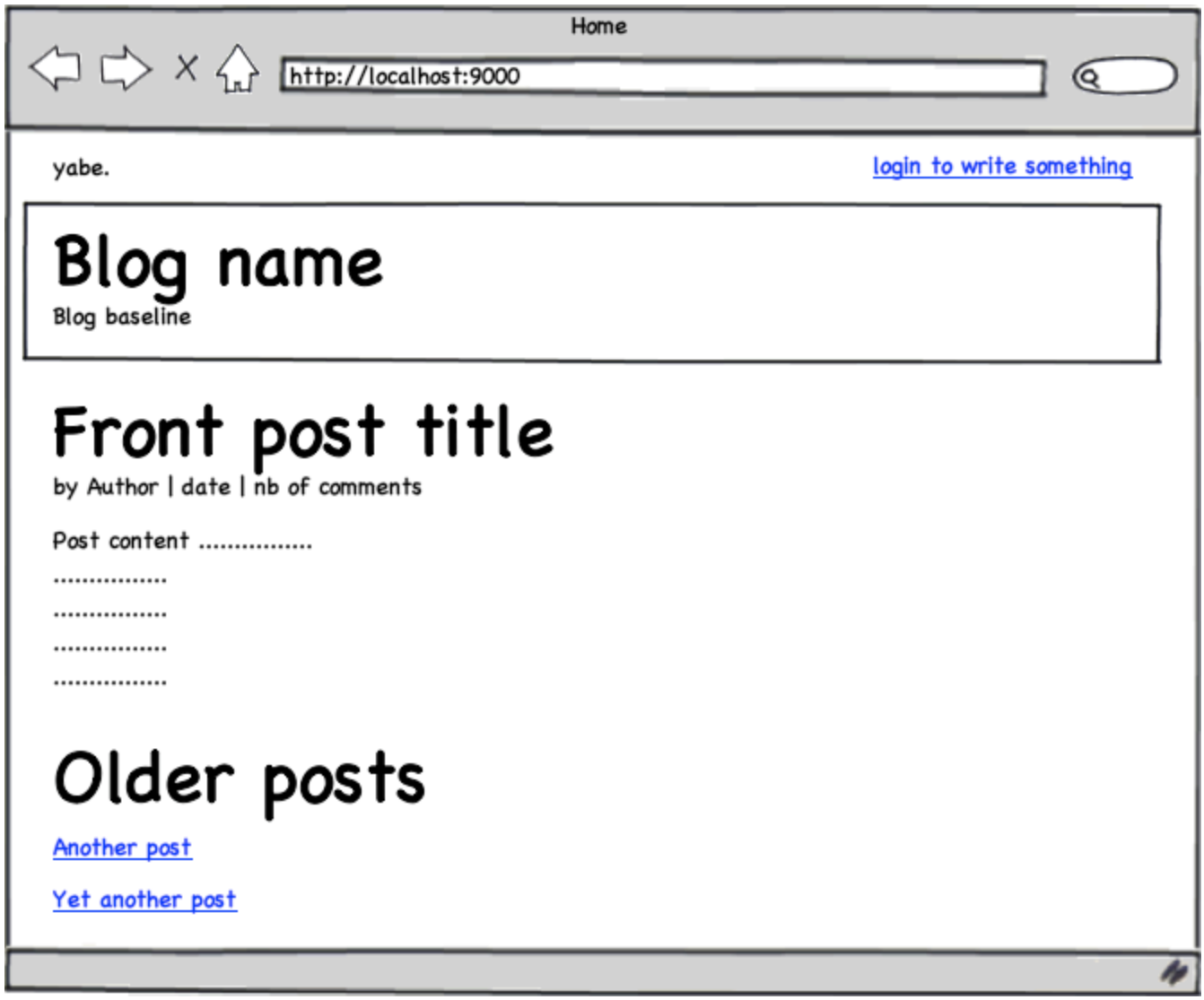
You can now commit your project.

```
$ bzr commit -m "The model layer is ready"
```

Go to the next part.

Now that we have built a first data model, it's time to start to create the first page of the application. This page will just show the most recent posts, as well as a list of older posts.

Here is a mock-up of what we want to achieve:



Bootstrapping with default data

In fact, before coding the first screen we need one more thing. Working on a web application without test data is not fun. You can't even test what you're doing. But because we haven't developed the contribution screens yet, we can't populate the blog with posts ourselves.

One way to inject default data into the blog is to load a fixture file at application load time. To do that we will create a Bootstrap Job. A Play job is something that executes itself outside of any HTTP request, for example at the application start or at specific interval using a CRON job.

Let's create the **/yabe/app/Bootstrap.java** job that will load a set of default data using **Fixtures**:

```
import play.*;
import play.jobs.*;
import play.test.*;

import models.*;

@OnApplicationStart
public class Bootstrap extends Job {

    public void doJob() {
        // Check if the database is empty
        if(User.count() == 0) {
            Fixtures.load("initial-data.yml");
        }
    }

}
```

We have annotated this Job with the **@OnApplicationStart** annotation to tell Play that we want to run this job synchronously at application start-up.

In fact this job will be run differently in DEV or PROD modes. In DEV mode, Play waits for a first request to start. So this job will be executed synchronously at the first request. That way, if the job fails, you will get the error message in your browser. In PROD mode however, the job will be executed at application start-up (synchrously with the **play run** command) and will prevent the application from starting in case of an error.

You have to create an **initial-data.yml** in the **yabe/conf/** directory. You can of course reuse the **data.yml** content that we just used for tests previously.

Now run the application using **play run** and display the page http://localhost:9000/ in the browser.

The blog home page

This time, we can really start to code the home page.

Do you remember how the first page is displayed? First the routes file specifies that the **/** URL will invoke the **controllers.Application.index()** action method. Then this method calls **render()** and executes the **/yabe/app/views/Application/index.html** template.

We will keep these components but add code to them to load the posts list and display them.

Open the **/yabe/app/controllers/Application.java** controller and modify the **index()** action to load the posts list, as is:

```
package controllers;

import java.util.*;

import play.*;
import play.mvc.*;

import models.*;

public class Application extends Controller {

    public static void index() {
        Post frontPost = Post.find("order by postedAt desc").first();
        List<Post> olderPosts = Post.find(
            "order by postedAt desc"
        ).from(1).fetch(10);
        render(frontPost, olderPosts);
    }

}
```

Can you see how we pass objects to the **render** method? It will allow us to access them from the template using the same name. In this case, the variables **frontPost** and **olderPosts** will be available in the template.

Open the **/yabe/app/views/Application/index.html** and modify it to display these objects:

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{if frontPost}
    <div class="post">
        <h2 class="post-title">
            <a href="#">${frontPost.title}</a>
        </h2>
        <div class="post-metadata">
            <span class="post-author">by
${frontPost.author.fullname}</span>
            <span class="post-date">${frontPost.postedAt.format('MMM
dd')}</span>
            <span class="post-comments">
                 | 
                ${frontPost.comments.size() ?: 'no'}
                comment${frontPost.comments.size().pluralize()}</a>
                #{if frontPost.comments}
                    , latest by ${frontPost.comments[0].author}
                #{/if}
            </span>
        </div>
        <div class="post-content">
            ${frontPost.content.nl2br()}
        </div>
    </div>

    #{if olderPosts.size() > 1}
        <div class="older-posts">
            <h3>Older posts <span class="from">from this
blog</span></h3>

            #{list items:olderPosts, as:'oldPost'}
                <div class="post">
                    <h2 class="post-title">
                        <a href="#">${oldPost.title}</a>
                    </h2>
                    <div class="post-metadata">
                        <span class="post-author">
                            by ${oldPost.author.fullname}
                        </span>
                        <span class="post-date">
                            ${oldPost.postedAt.format('dd MMM yy')}
                        </span>
                        <div class="post-comments">
                            ${oldPost.comments.size() ?: 'no'}
                            comment${oldPost.comments.size().pluralize()}
                            #{if oldPost.comments}
                                - latest by ${oldPost.comments[0].author}
                            #{/if}
                        </div>
                    </div>
                </div>
            #{/list}
        </div>

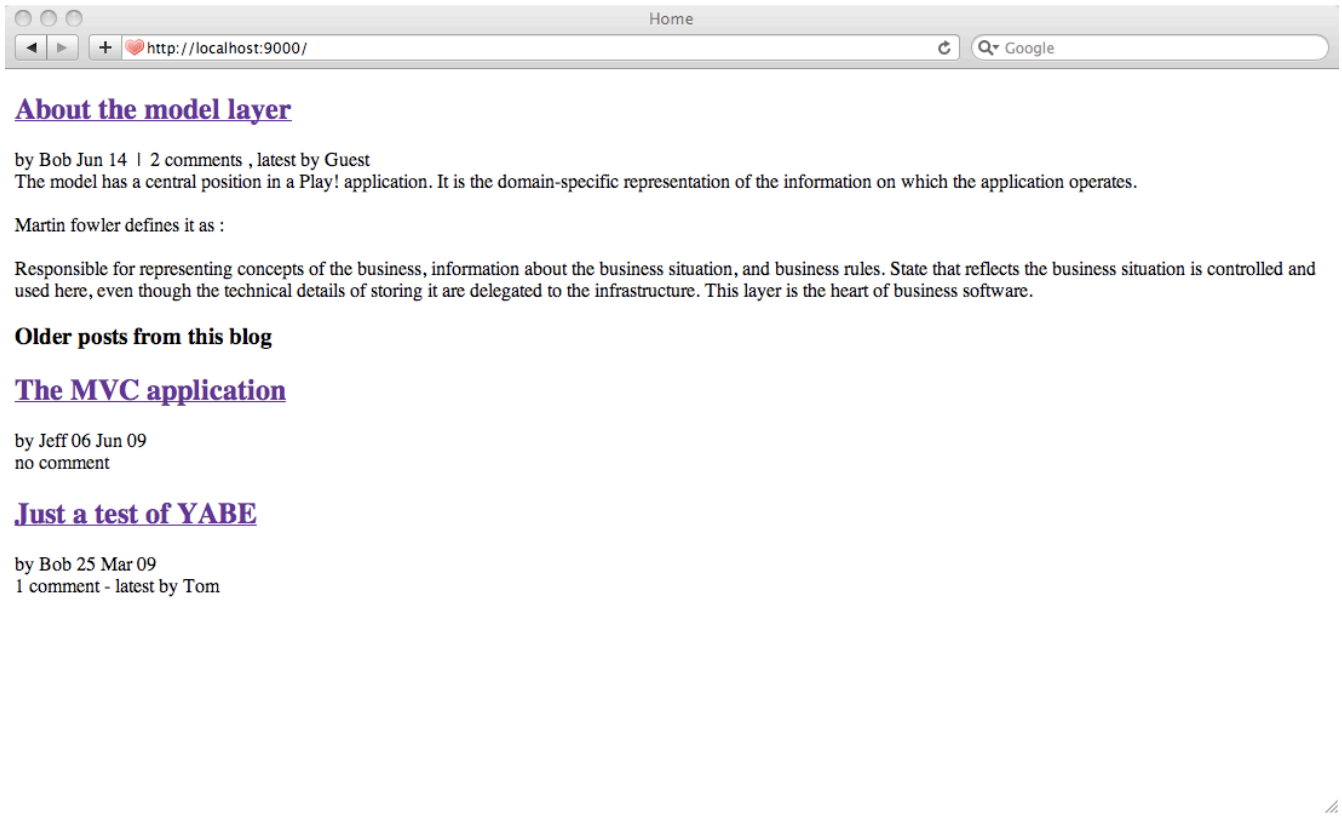
    #{/if}

#{/if}

#{else}
    <div class="empty">
        There is currently nothing to read here.
    </div>
#{/else}
```

You can read about this template works in the Templates chapter. Basically, it allows you to access your Java objects dynamically. Under the hood we use Groovy. Most of the pretty constructs you see (like the **?:** operator) come from Groovy. But you don't really need to learn Groovy to write Play templates. If you're already familiar with another template language like JSP with JSTL you won't be lost.

OK, now refresh the blog home page.



Not pretty but it works!

However you can see we have already started to duplicate code. Because we will display posts in several ways (full, full with comment, teaser) we should create something like a function that we could call from several templates. This is exactly what a Play tag does!

To create a tag, just create the new **/yabe/app/views/tags/display.html** file. A tag is just another template. It has parameters (like a function). The **#{display /}** tag will have two parameters: the Post object to display and the display mode which will be one of 'home', 'teaser' or 'full'.

```
*{ Display a post in one of these modes: 'full', 'home' or 'teaser' }*

<div class="post ${_as == 'teaser' ? 'teaser' : ''}">
    <h2 class="post-title">
        <a href="#">${_post.title}</a>
    </h2>
    <div class="post-metadata">
        <span class="post-author">by ${_post.author.fullname}</span>,
        <span class="post-date">${_post.postedAt.format('dd MMM
yy')}</span>
        #{if _as != 'full'}
            <span class="post-comments">
                 |  ${_post.comments.size() ?: 'no'}
                comment${_post.comments.size().pluralize()}
                #{if _post.comments}
                    , latest by ${_post.comments[0].author}
                #{/if}
            </span>
        #{/if}
    </div>
    #{if _as != 'teaser'}
        <div class="post-content">
            <div class="about">Detail: </div>
            ${_post.content.nl2br()}
        </div>
    #{/if}
</div>

#{if _as == 'full'}
    <div class="comments">
        <h3>
            ${_post.comments.size() ?: 'no'}
            comment${_post.comments.size().pluralize()}
        </h3>

        #{list items:_post.comments, as:'comment'}
            <div class="comment">
                <div class="comment-metadata">
                    <span class="comment-author">by
${comment.author},</span>
                    <span class="comment-date">
                        ${comment.postedAt.format('dd MMM yy')}
                    </span>
                </div>
                <div class="comment-content">
                    <div class="about">Detail: </div>
                    ${comment.content.escape().nl2br()}
                </div>
            </div>
        #{/list}

    </div>
#{/if}
```

Now using this tag we can rewrite the home page without code duplication:

```
#{extends 'main.html' /}
#{set title:'Home' /}

#{if frontPost}

    #{display post:frontPost, as:'home' /}

    #{if olderPosts.size()}

        <div class="older-posts">
            <h3>Older posts <span class="from">from this
blog</span></h3>

            #{list items:olderPosts, as:'oldPost'}
                #{display post:oldPost, as:'teaser' /}
            #{/list}
        </div>

    #{/if}

#{/if}

#{else}
    <div class="empty">
        There is currently nothing to read here.
    </div>
#{/else}
```

Reload the page and check that all is fine.

Improving the layout

As you can see, the **index.html** template extends **main.html**. Because we want to provide a common layout for all blog pages, with the blog title and authentication links, we need to modify this file.

Edit the **/yabe/app/views/main.html** file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        <link rel="stylesheet" type="text/css" media="screen"
            href="@{'/public/stylesheets/main.css'}" />
        <link rel="shortcut icon" type="image/png"

href="@{'/public/file:///Users/nicolasleroux/Projects/Personal/1.1/documentation/images/
/>
    </head>
    <body>

        <div id="header">
            <div id="logo">
                yabe.
            </div>
            <ul id="tools">
                <li>
                    <a href="#">Log in to write something</a>
                </li>
            </ul>
            <div id="title">
                <span class="about">About this blog</span>
                <h1><a href="#">${blogTitle}</a></h1>
                <h2>${blogBaseline}</h2>
            </div>
        </div>

        <div id="main">
            #{doLayout /}
        </div>

        <p id="footer">
            Yabe is a (not that) powerful blog engine built with the
            <a href="http://www.playframework.org">Play framework</a>
            as a tutorial application.
        </p>

    </body>
</html>
```

Refresh and check the result. It seems to work, except that the **blogTitle** and the **blogBaseLine** variables are not displayed. This is because we didn't pass them to the **render(…)** call. Of course we could add them to the **render()** call in the **index** action. But because the **main.html** file will be used as main template for all application actions, we don't want to add them every time.

One way to execute the same code for each action of a controller (or a hierarchy of controllers) is to define a **@Before** interceptor.

Let's add the **addDefaults()** method to the Application controller:

```
@Before
static void addDefaults() {
    renderArgs.put("blogTitle", Play.configuration.getProperty("blog.title"));
    renderArgs.put("blogBaseline", Play.configuration.getProperty("blog.baseline"));
}
```

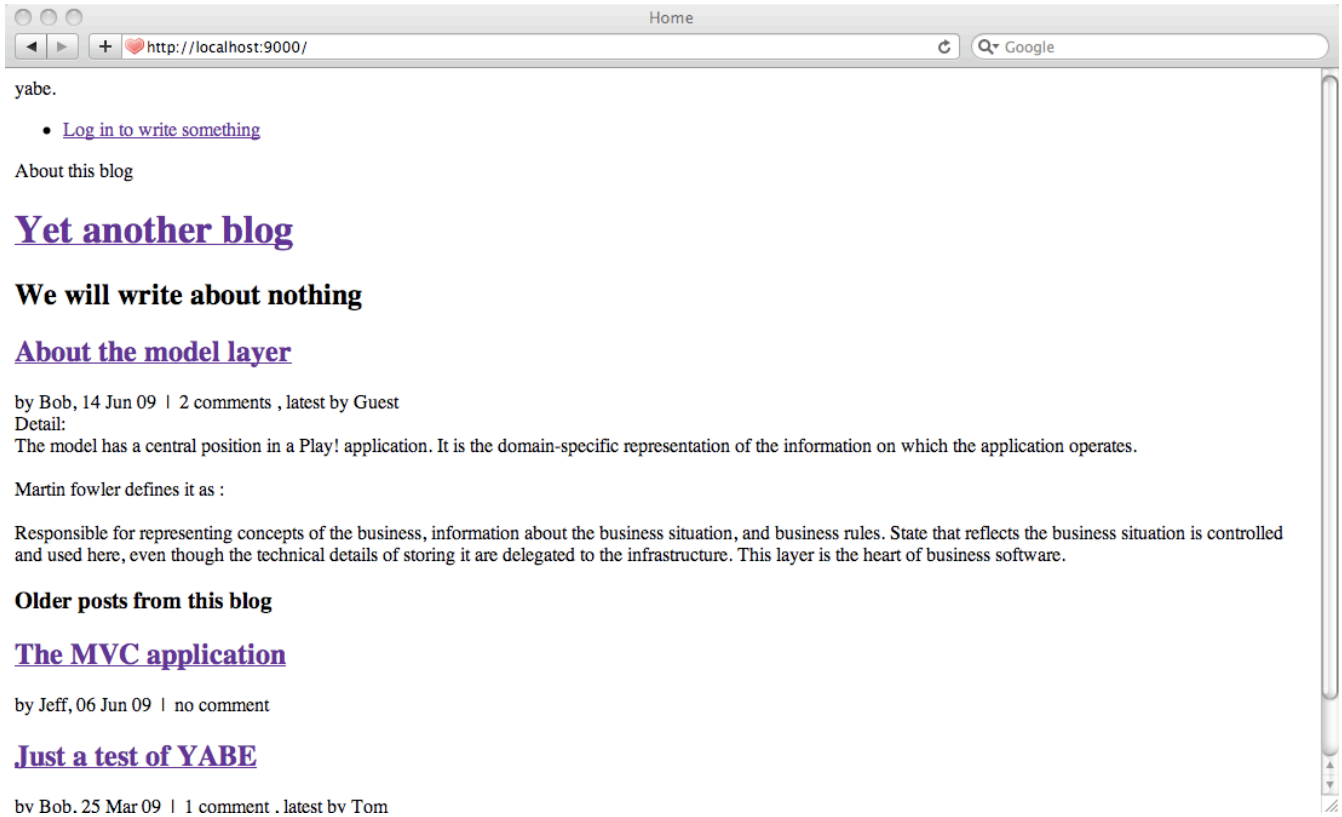
You will need to import **play.Play** in the **Application.java** file.

All variables added to the **renderArgs** scope will be available from the templates. And you can see that the method reads the variable's values from the **Play.configuration** object. This object contains all configuration keys from the **/yabe/conf/application.conf** file.

Add these two keys to the configuration file:

```
# Blog engine configuration
# ~~~~~
blog.title=Yet another blog
blog.baseline=We won't write about anything
```

Reload the home page and check that it works!



Adding some style

Now the blog home page is almost done, but it's not very pretty. We'll add some style to make it shinier. As you have seen, the main template file main.html includes the **/public/stylesheets/main.css** stylesheet. We'll keep it but add more style rules to it.

You can download it here, and copy it to the **/public/stylesheets/main.css** file.

Refresh the home page and you should now see a styled page.



Commit your work

The blog home page is now finished. As usual we can commit this blog version to bazaar:

```
$ bzr st
$ bzr add
$ bzr commit -m 'Home page'
```

Go to the next part.

The blog home page is now set, and we will continue by writing the post details page. This page will show all the comments about the current post, and will include a form to post new comments.

Creating the 'show' action

To display the post details page, we will need a new action method on the **Application** controller. Let's call it **show()**:

```
public static void show(Long id) {
    Post post = Post.findById(id);
    render(post);
}
```

As you can see this action is pretty simple. We declare the **id** method parameter to automatically retrieve the HTTP **id** parameter as a **Long** Java object. This parameter will be extracted either from the query string, from the URL path or from the request body.

If we try to send an **id** HTTP parameter that is not a valid number, the **id** variable value will be **null** and Play will automatically add a validation error to the **errors** container.

This action will display the **/yabe/app/views/Application/show.html** template:

```
#{extends 'main.html' /}
#{set title:post.title /}

#{display post:post, as:'full' /}
```

Because we've already written the **display** tag, this page is really simple to write.

Adding links to the details page

In the display tag we've left all links empty (using #). It's now time to make these links point to the **Application.show** action. With Play you can easily build links in a template using the **@{...} notation**. This syntax uses the router to 'reverse' the URL needed to call the specified action.

Let's edit the **/yabe/app/views/tags/display.html** tag:

```
...
<h2 class="post-title">
    <a href="@{Application.show(_post.id)}">${_post.title}</a>
</h2>
...
```

You can new refresh the home page, and click a post title to display the post.



It's great, but it lacks a link to go back to the home page. Edit the **/yabe/app/views/main.html** template to complete the title link:

```
...
<div id="title">
    <span class="about">About this blog</span>
    <h1><a href="@{Application.index()}">${blogTitle}</a></h1>
    <h2>${blogBaseline}</h2>
</div>
...
```

We can now navigate between the home page and the post detail pages.

Specifying a better URL

As you can see, the post detail page URL looks like:

```
/application/show?id=1
```

This is because Play has used the default 'catch all' route.

```
*       /{controller}/{action}                  {controller}.{action}
```

We can have a better URL by specifying a custom path for the **Application.show** action. Edit the **/yabe/conf/routes** file and add this route after the first one:

```
GET     /posts/{id}                             Application.show
```

This way the **id** parameter will be extracted from the URL path. You can read more about URI patterns on the manual page about Route File Syntax.

Refresh the browser and check that it now uses the correct URL.

Adding pagination

To allow users to navigate easily through posts, we will add a pagination mechanism. We'll extend the Post class to be able to fetch previous and next post as required:

```
public Post previous() {
    return Post.find("postedAt < ? order by postedAt desc", postedAt).first();
}

public Post next() {
    return Post.find("postedAt > ? order by postedAt asc", postedAt).first();
}
```

We will call these methods several times during a request so they could be optimized, but they're good enough for now. Also, add the pagination links at the top of the **show.html** template (before the **#{display/}** tag):

```
<ul id="pagination">
    #{if post.previous()}
        <li id="previous">
            <a href="@{Application.show(post.previous().id)}">
                ${post.previous().title}
            </a>
        </li>
    #{/if}
    #{if post.next()}
        <li id="next">
            <a href="@{Application.show(post.next().id)}">
                ${post.next().title}
            </a>
        </li>
    #{/if}
</ul>
```

It's better now.

Adding the comment form

Now it's time to set up a comments form. We'll start by adding the **postComment** action method to the Application controller.

```
public static void postComment(Long postId, String author, String content) {
    Post post = Post.findById(postId);
    post.addComment(author, content);
    show(postId);
}
```

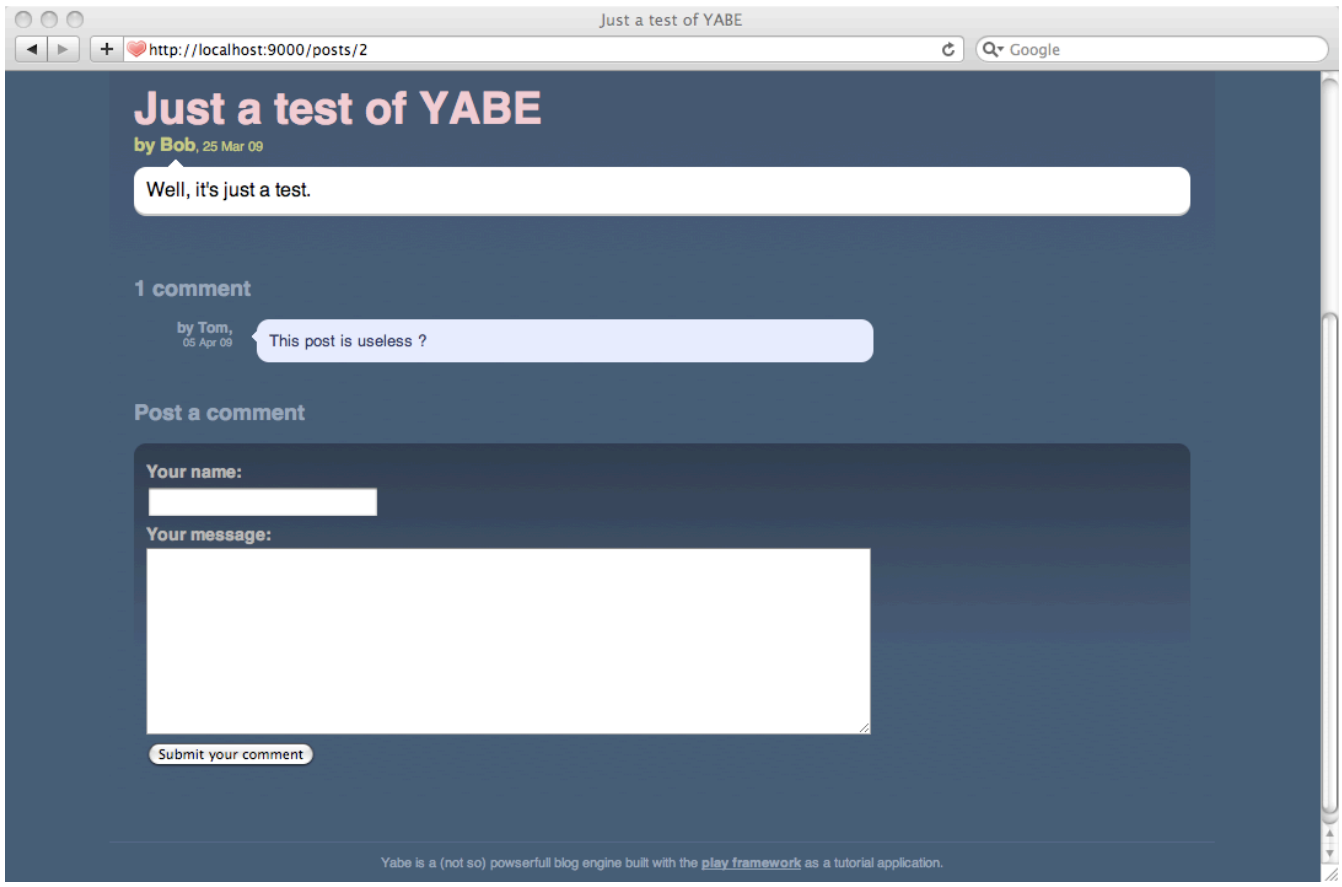
As you can see we just reuse the **addComment()** method we previously added to the Post class.

Let's write the HTML form in the **show.html** template (after the **#{display /}** tag in fact):

```
<h3>Post a comment</h3>

#{form @Application.postComment(post.id)}
    <p>
        <label for="author">Your name: </label>
        <input type="text" name="author" id="author" />
    </p>
    <p>
        <label for="content">Your message: </label>
        <textarea name="content" id="content"></textarea>
    </p>
    <p>
        <input type="submit" value="Submit your comment" />
    </p>
#{/form}
```

You can now try posting a new comment. It should just work.



Adding validation

Currently we don't validate the form content before creating the comment. We would like to make both fields required. We can easily use the Play validation mechanism to ensure that the HTTP parameters are correctly filled in. Modify the **postComment** action to add **@Required** validation annotations and check that no error occurs:

```
public static void postComment(Long postId, @Required String author, @Required String
content) {
    Post post = Post.findById(postId);
    if (validation.hasErrors()) {
        render("Application/show.html", post);
    }
    post.addComment(author, content);
    show(postId);
}
```

Don't forget to import **play.data.validation.*** as well.

As you can see, in case of validation errors, we re-display the post detail page. We have to modify the form code to display the error message:

```
<h3>Post a comment</h3>

#{form @Application.postComment(post.id)}

    #{ifErrors}
        <p class="error">
            All fields are required!
        </p>
    #{/ifErrors}

    <p>
        <label for="author">Your name: </label>
        <input type="text" name="author" id="author" value="${params.author}" />
    </p>
    <p>
        <label for="content">Your message: </label>
        <textarea name="content" id="content">${params.content}</textarea>
    </p>
    <p>
        <input type="submit" value="Submit your comment" />
    </p>
#{/form}
```

Note that we reuse the posted parameters to fill the HTML input values.

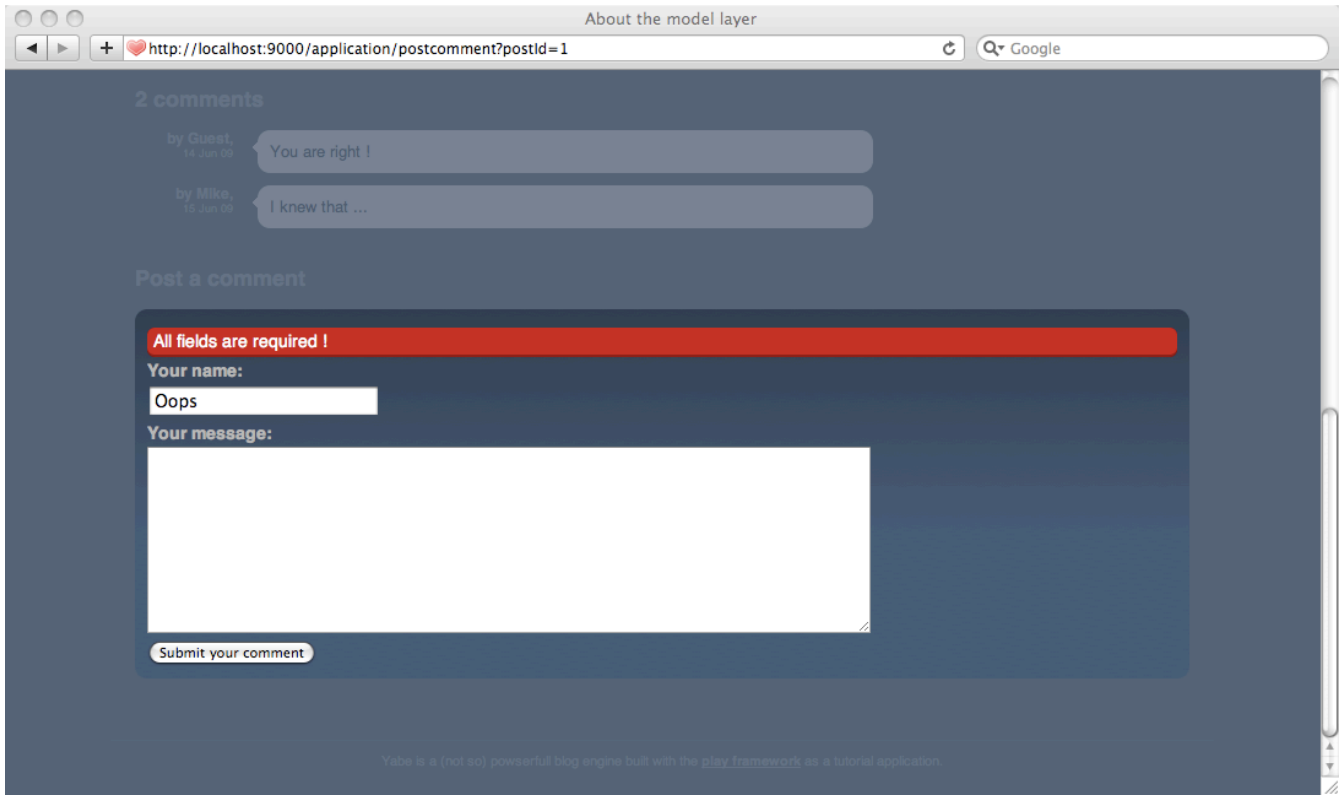
To make the UI feedback more pleasant for the poster, we will add a little JavaScript to automatically set focus on the comment form in case of an error. As this script uses JQuery and JQuery Tools as support libraries, you have to include them. Download these two libraries to the **yabe/public/javascripts/** directory and modify the **main.html** template to include them:

```
...
    <script src="@{'/public/javascripts/jquery-1.3.2.min.js'}"></script>
    <script src="@{'/public/javascripts/jquery.tools.min.js'}"></script>
</head>
```

Now you can add this script to the **show.html** template (add it at the end of the page):

```
<script type="text/javascript" charset="utf-8">
    $(function() {
        // Expose the form
        $('form').click(function() {
            $('form').expose({api: true}).load();
        });

        // If there is an error, focus to form
        if($('form .error').size()) {
            $('form').expose({api: true, loadSpeed: 0}).load();
            $('form input').get(0).focus();
        }
    });
</script>
```



The comment form looks pretty cool now. We will add two more things.

First, we will display a success message after a comment is successfully posted. For that, we use the flash scope that allows us to pass messages from one action call to the next one.

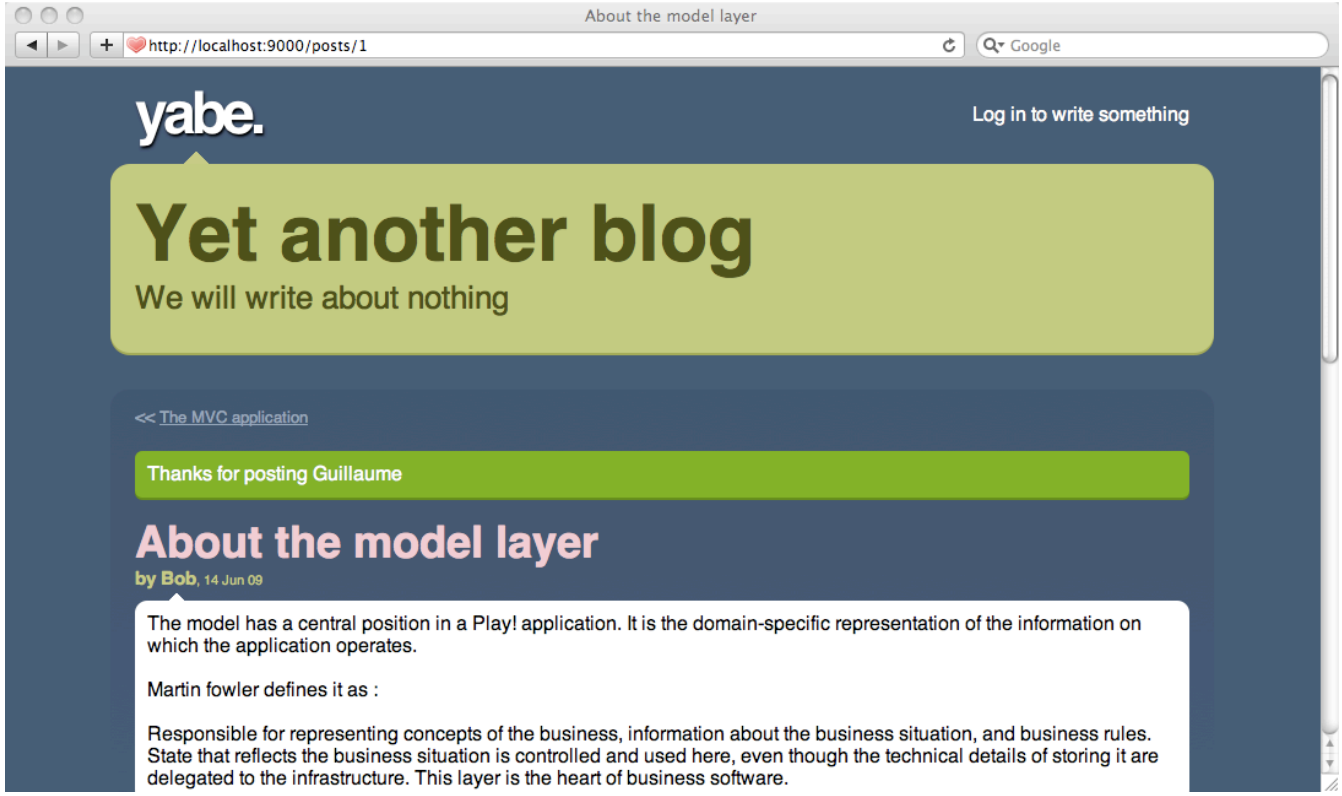
Modify the **postComment** action to add a success message:

```
public static void postComment(Long postId, @Required String author, @Required String
content) {
    Post post = Post.findById(postId);
    if(validation.hasErrors()) {
        render("Application/show.html", post);
    }
    post.addComment(author, content);
    flash.success("Thanks for posting %s", author);
    show(postId);
}
```

and display the success message in **show.html** if present (add it at the top the page):

```
...
#{if flash.success}
    <p class="success">${flash.success}</p>
#{/if}

#{display post:post, as:'full' /}
...
```



The last thing we will adjust in this form is the URL used for the **postComment** action. As always, it uses the default catch-all route because we didn't define any specific route. So add this route to the application routes file:

```
POST    /posts/{postId}/comments                Application.postComment
```

That's done. As always, commit the version to bazaar.

Go to the next part.

Because anyone can post a comment to our blog engine, we should protect it a little to avoid automated spam. A simple way to protect a form from this is to add a captcha image.

Generating the captcha image

We'll start to see how we can easily generate a captcha image using Play. Basically we will just use another action, except that it will return a binary stream instead of the HTML responses like we've returned so far.

Since Play is a **full-stack** web framework, we try to include built-in constructs for web applications' most typical needs; generating a captcha is one of them. We can use the **play.libs.Images** utility to simply generate a captcha image, and then write it to the HTTP response.

As usual, we will start with a simple implementation. Add the **captcha** action to the **Application** controller:

```
public static void captcha() {
    Images.Captcha captcha = Images.captcha();
    renderBinary(captcha);
}
```

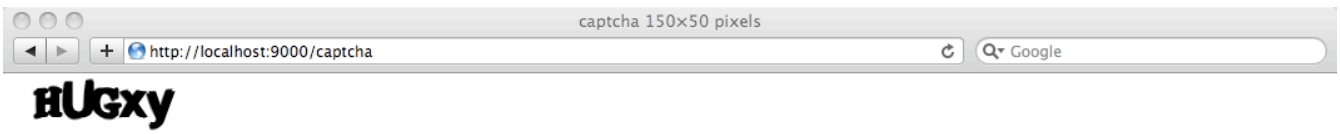
Note that we can pass the captcha object directly to the renderBinary() method because the Images.Captcha class implements java.io.InputStream.

Don't forget to import **play.libs.***.

Now add a new route to the **/yabe/conf/routes** file:

```
GET     /captcha                                Application.captcha
```

And try the **captcha** action by opening http://localhost:9000/captcha.



It should generate a random text for each refresh.

How do we manage the state?

Until now it was easy, but the most complicated part is coming. To validate the captcha we need to save the random text written to the captcha image somewhere and then check it at the form submission time.

Of course we could just put the text to the user session at the image generation time and then retrieve it later. But this solution has two drawbacks:

First, the Play session is stored as a cookie. It solves a lot of problems in terms of architecture but has a lot of implications. Data written to the session cookie are signed (so the user can't modify them) but not encrypted. If we write the captcha code to the session anybody could easily resolve it by reading the session cookie.

Second, remember that Play is a **stateless** framework. We want to manage things in a purely stateless way. Typically, what happens if a user simultaneously opens two different blog pages with two different captcha images? We have to track the captcha code for each form.

So to resolve the problem we need two things. We will store the captcha secret key on the server side. Because it is transient data we can easily use the Play **Cache**. Moreover because cached data have a limited life time it will add one more security mechanism (let's say that a captcha code will be available for only 10mn). Then to resolve the code later we need to generate a **unique ID**. This unique ID will be added to each form as an hidden field and implicitely references a generated captcha code.

This way we elegantly solve our state problem.

Modify the **captcha** action as is:

```
public static void captcha(String id) {
    Images.Captcha captcha = Images.captcha();
    String code = captcha.getText("#E4EAFD");
    Cache.set(id, code, "10mn");
    renderBinary(captcha);
}
```

Note that the **getText()** method takes any color as parameter. It will use this color to draw the text.

Don't forget to import **play.cache.***.

Adding the captcha image to the comment form

Now, before displaying a comment form we will generate a unique ID. Then we will modify the HTML form to integrate a captcha image using this ID, and add the ID to another hidden field.

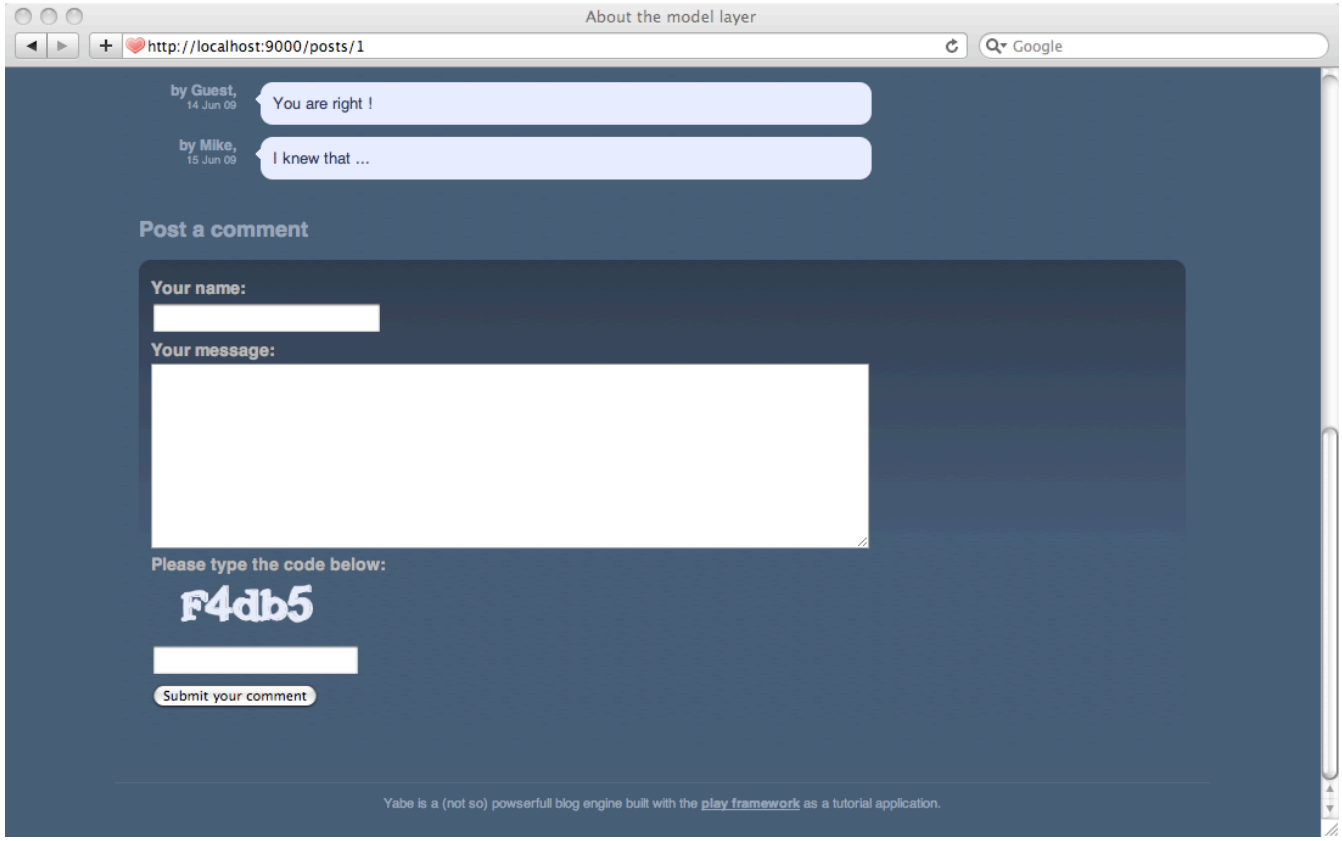
Let's rewrite the **Application.show** action:

```
public static void show(Long id) {
    Post post = Post.findById(id);
    String randomID = Codec.UUID();
    render(post, randomID);
}
```

And now the form in the **/yable/app/views/Application/show.html** template:

```
...
<p>
    <label for="content">Your message: </label>
    <textarea name="content" id="content">${params.content}</textarea>
</p>
<p>
    <label for="code">Please type the code below: </label>
    <img src="@{Application.captcha(randomID)}" />
    <br />
    <input type="text" name="code" id="code" size="18" value="" />
    <input type="hidden" name="randomID" value="${randomID}" />
</p>
<p>
    <input type="submit" value="Submit your comment" />
</p>
...
```

Good start. The comment form now has a captcha image.



Validating the captcha

Now we just have to validate the captcha. We have added the **randomID** as an hidden field right? So we can retrieve it in the **postComment** action, then retrieve the actual code from Cache and finally compare it to the submitted code.

Not that difficult. Let's modify the **postComment** action.

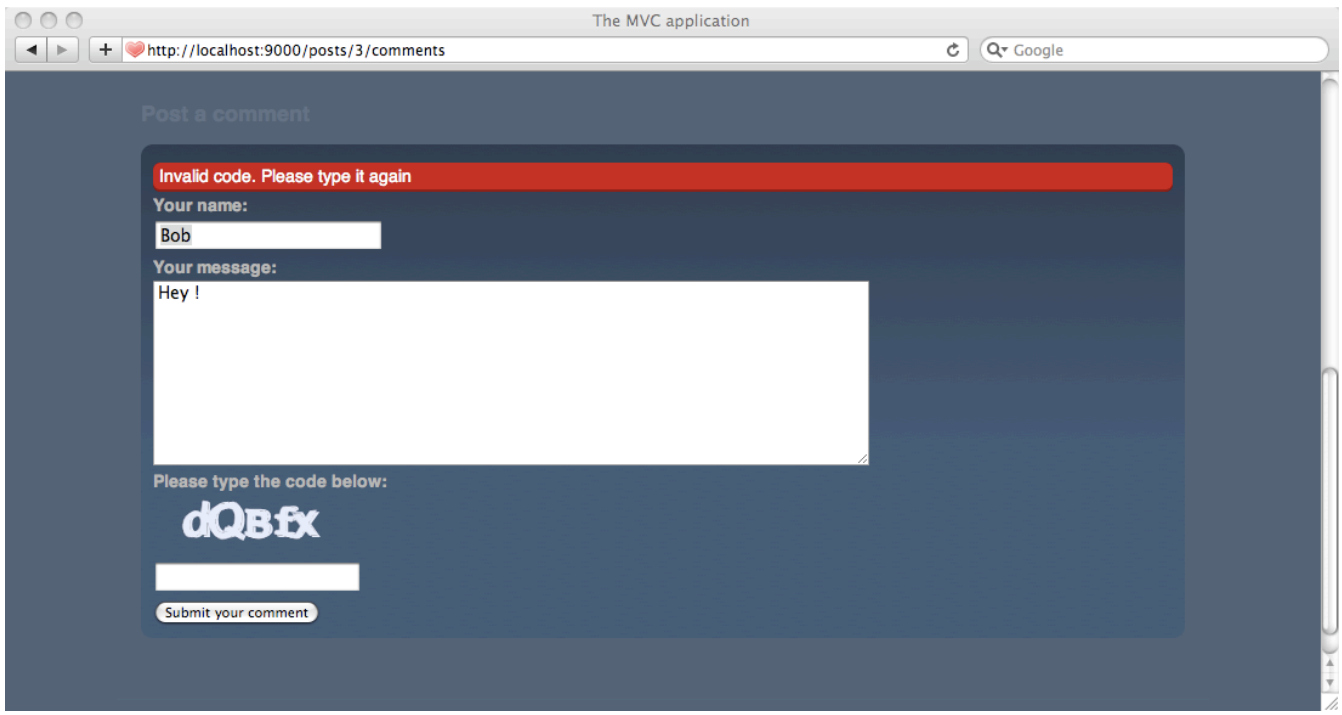
```
public static void postComment(
        Long postId,
        @Required(message="Author is required") String author,
        @Required(message="A message is required") String content,
        @Required(message="Please type the code") String code,
        String randomID)
{
    Post post = Post.findById(postId);
    validation.equals(
        code, Cache.get(randomID)
    ).message("Invalid code. Please type it again");
    if(validation.hasErrors()) {
        render("Application/show.html", post, randomID);
    }
    post.addComment(author, content);
    flash.success("Thanks for posting %s", author);
    Cache.delete(randomID);
    show(postId);
}
```

Because we now have more error messages, modify the way we display errors in the **show.html** template (yes we will just display the first error, it's good enough):

```
..
#{ifErrors}
    <p class="error">
        ${errors[0]}
    </p>
#{/ifErrors}
...
```

Typically for more complex forms, error messages are not managed this way but externalized in the **messages** file and each error is printed inside the corresponding field.

Check that the captcha is now fully functional.



Great!

Go to the next part.

As our blog will contain more posts, it will become more and more difficult to retrieve them. To help classify posts by subjects we will add tagging support.

The Tag model object

We will add one more object to the blog model definition. The Tag class itself is indeed very simple:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;

@Entity
public class Tag extends Model implements Comparable<Tag> {

    public String name;

    private Tag(String name) {
        this.name = name;
    }

    public String toString() {
        return name;
    }

    public int compareTo(Tag otherTag) {
        return name.compareTo(otherTag.name);
    }

}
```

Because we want something like lazy tag creation we will always get them using the **findOrCreateByName(String name)** factory method. Let's add it to the Tag class:

```
public static Tag findOrCreateByName(String name) {
    Tag tag = Tag.find("byName", name).first();
    if(tag == null) {
        tag = new Tag(name);
    }
    return tag;
}
```

Tagging posts

Now it's time to link the new Tag model with the Post one. Let's add the correct relationship to the Post class:

```
...
@ManyToMany(cascade=CascadeType.PERSIST)
public Set<Tag> tags;

public Post(User author, String title, String content) {
    this.comments = new ArrayList<Comment>();
    this.tags = new TreeSet<Tag>();
    this.author = author;
    this.title = title;
    this.content = content;
    this.postedAt = new Date();
}
...
```

Note that we use a **TreeSet** here in order to keep the tag list in a predictable order (alphabetical order in fact, because of our previous compareTo implementation).

We will keep this relationship unidirectional.

We will also add a bunch of helper methods to make tag management simpler. First one, the ability to tag a Post:

```
...
public Post tagItWith(String name) {
    tags.add(Tag.findOrCreateByName(name));
    return this;
}
...
```

Then the ability to retrieve all posts tagged with a specific tag:

```
...
public static List<Post> findTaggedWith(String tag) {
    return Post.find(
        "select distinct p from Post p join p.tags as t where t.name = ?", tag
    ).fetch();
}
...
```

It is time to write a new test case to test this stuff. Restart the server in **test** mode by typing:

```
$ play test
```

And add a new **@Test** to the **BasicTest** class:

```
@Test
public void testTags() {
    // Create a new user and save it
    User bob = new User("bob@gmail.com", "secret", "Bob").save();

    // Create a new post
    Post bobPost = new Post(bob, "My first post", "Hello world").save();
    Post anotherBobPost = new Post(bob, "Hop", "Hello world").save();

    // Well
    assertEquals(0, Post.findTaggedWith("Red").size());

    // Tag it now
    bobPost.tagItWith("Red").tagItWith("Blue").save();
    anotherBobPost.tagItWith("Red").tagItWith("Green").save();

    // Check
    assertEquals(2, Post.findTaggedWith("Red").size());
    assertEquals(1, Post.findTaggedWith("Blue").size());
    assertEquals(1, Post.findTaggedWith("Green").size());

}
```

Make sure that it works.

A little more difficult now

Well, we won't use it in our blog right now, but what if we wanted to retrieve posts tagged with several tags? It's more difficult than it seems.

I give you the needed JPQL query because you will likely use it in several web projects:

```
...
public static List<Post> findTaggedWith(String... tags) {
    return Post.find(
            "select distinct p from Post p join p.tags as t where t.name in (:tags)
group by p.id, p.author, p.title, p.content,p.postedAt having count(t.id) = :size"
    ).bind("tags", tags).bind("size", tags.length).fetch();
}
...
```

The tricky part is that we have to use a **having count** statement to filter only posts that have exactly **all tags** from the joined view.

Note that we can't use the **Post.find("...", tags, tags.count)** signature here. It's just because **tags** is already a **vararg**.

You can test it by adding more checks to the previous test:

```
...
assertEquals(1, Post.findTaggedWith("Red", "Blue").size());
assertEquals(1, Post.findTaggedWith("Red", "Green").size());
assertEquals(0, Post.findTaggedWith("Red", "Green", "Blue").size());
assertEquals(0, Post.findTaggedWith("Green", "Blue").size());
...
```

The tag cloud

Where we have tags, we need a tag cloud. Let's add a method to the **Tag** class to generate the tag cloud:

```
public static List<Map> getCloud() {
    List<Map> result = Tag.find(
        "select new map(t.name as tag, count(p.id) as pound) from Post p join p.tags
as t group by t.name order by t.name"
    ).fetch();
    return result;
}
```

Here we use a handy Hibernate feature that allows to return a custom object from a JPA query. It will result a **List** containing for each tag a **Map** with two attributes: **name** for the tag name and **pound** for the tag count.

Let's test it by adding one more check to our tags test:

```
...
List<Map> cloud = Tag.getCloud();
assertEquals(
    "[{tag=Blue, pound=1}, {tag=Green, pound=1}, {tag=Red, pound=2}]",
    cloud.toString()
);
```

Adding tags to the Blog UI

We can now use the new tagging stuff to add one more way to browse the blog. As always, to work efficiently we need to add a bunch of test tags to our initial data set.

Modify the **/yabe/conf/initial-data.yml** file to add some tags to the tests posts. For example:

```
...
Tag(play):
    name:           Play

Tag(architecture):
    name:           Architecture

Tag(test):
    name:           Test

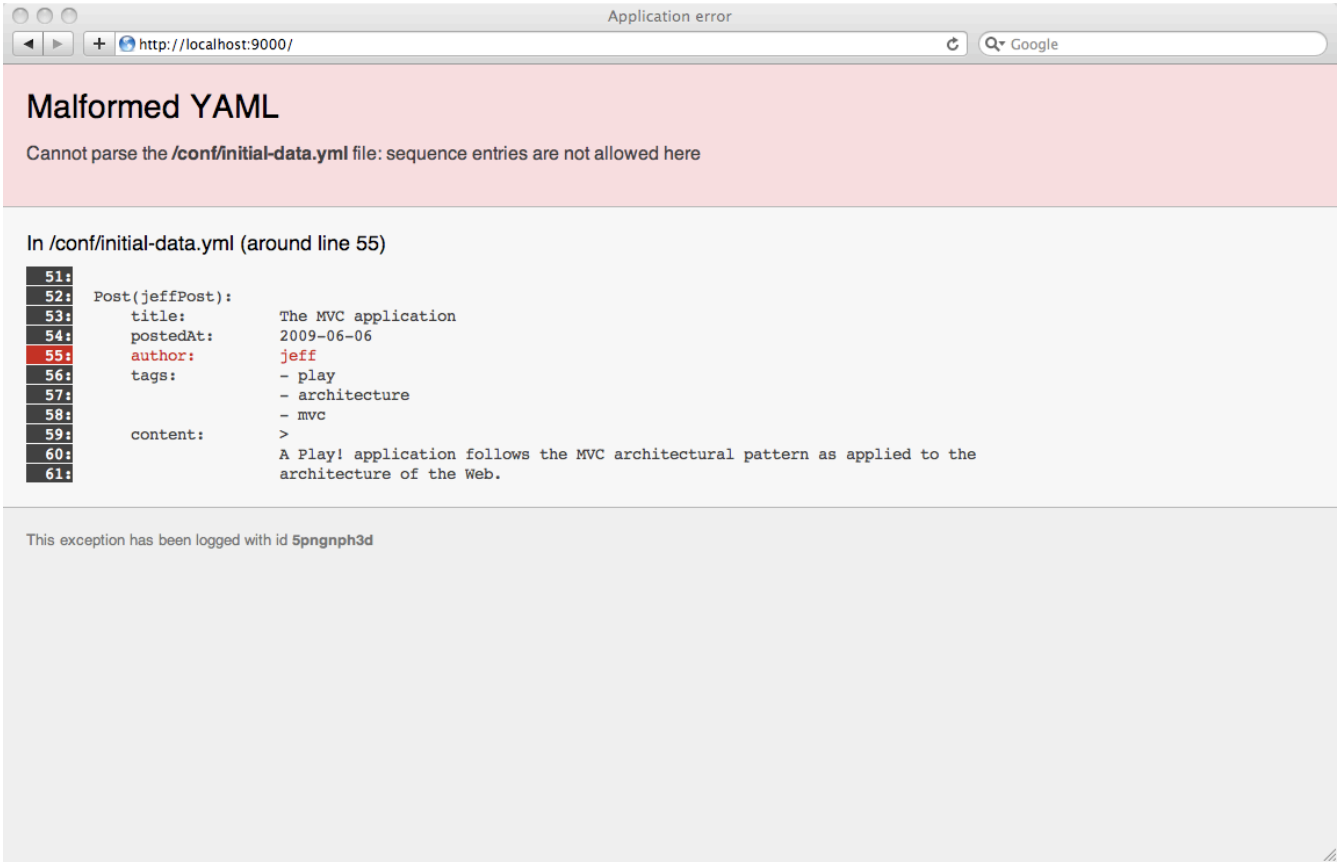
Tag(mvc):
    name:           MVC
...
```

And then add them to posts declaration as is:

```
...
Post(jeffPost):
    title:          The MVC application
    postedAt:       2009-06-06
    author:         jeff
    tags:
                    - play
                    - architecture
                    - mvc
    content:        >
                    A Play
...
```

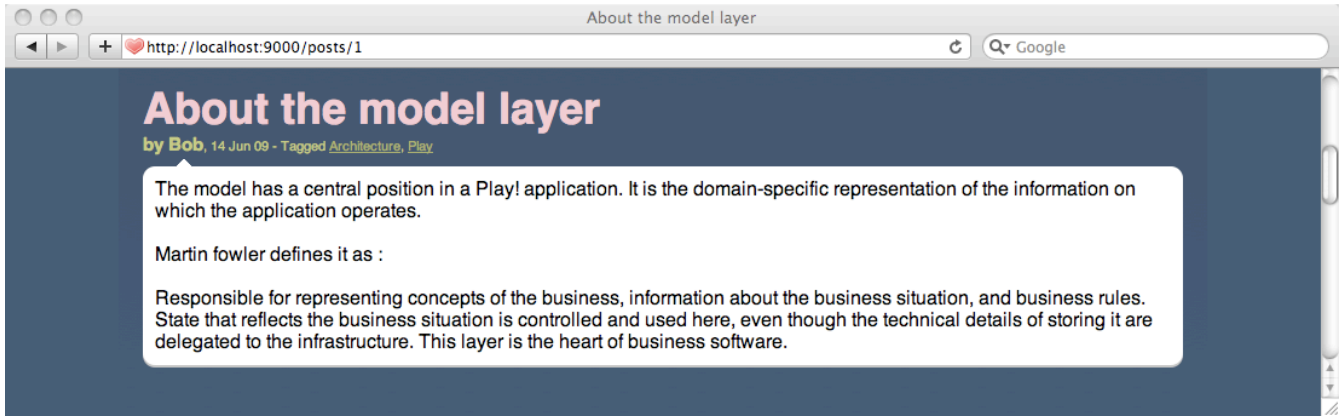
Add the Tags declaration at the top of the YAML file, because they need to be created before any Post references them.

You need to restart your application to force loading of the new initial data set. Note how Play even tells you about problems in YAML files:



Then modify the **#{display /}** tag to show the tag set on the **full** post view. Edit the **/yabe/app/views/tags/display.html** file:

```
...
#{if _as != 'full'}
    <span class="post-comments">
         |  ${_post.comments.size() ?: 'no'}
        comment${_post.comments.size().pluralize()}
        #{if _post.comments}
            , latest by ${_post.comments[0].author}
        #{/if}
    </span>
#{/if}
#{elseif _post.tags}
    <span class="post-tags">
        - Tagged
        #{list items:_post.tags, as:'tag'}
            <a href="#">${tag}</a>${tag_isLast ? '' : ', '}
        #{/list}
    </span>
#{/elseif}
...
```



The new 'tagged with' page

Now we can add a new way to list blog posts by tags. Above we left the link blank; we will replace it by a link to the new **listTagged** action:

```
...
- Tagged
#{list items:_post.tags, as:'tag'}
    <a href="@{Application.listTagged(tag.name)}">${tag}</a>${tag_isLast ?
'' : ', '}
#{/list}
...
```

And create the action method on the **Application** controller:

```
...
public static void listTagged(String tag) {
    List<Post> posts = Post.findTaggedWith(tag);
    render(tag, posts);
}
...
```

As always we create a specific route to keep clean URI:

```
GET     /posts/{tag}                    Application.listTagged
```

Well, we have a problem because we have an existing route that conflicts with this new one. These two routes will match the same URI:

```
GET     /posts/{id}                     Application.show
GET     /posts/{tag}                    Application.listTagged
```

But because we'll assume that an **id** is numeric and a **tag** is not, we can easily solve the situation using a regular expression to restrict the first route:

```
GET     /posts/{<[0-9]+>id}             Application.show
GET     /posts/{tag}                    Application.listTagged
```

Finally we just have to create the **/yabe/app/views/Application/listTagged.html** template that will be used by the new **listTagged** action:

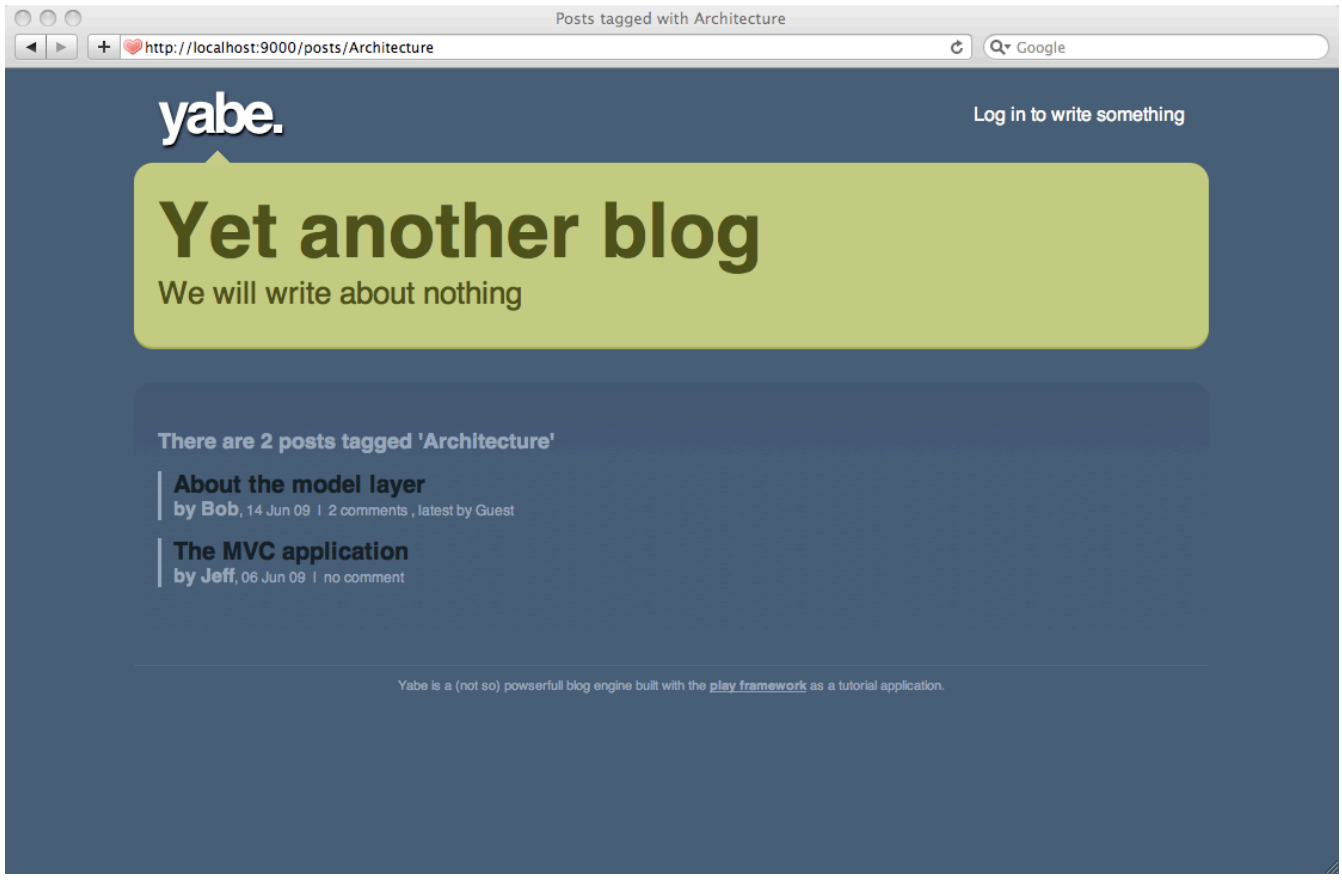
```
#{extends 'main.html' /}
#{set title:'Posts tagged with ' + tag /}

*{********* Title ********* }*

#{if posts.size() > 1}
   <h3>There are ${posts.size()} posts tagged '${tag}'</h3>
#{/if}
#{elseif posts}
    <h3>There is 1 post tagged '${tag}'</h3>
#{/elseif}
#{else}
    <h3>No post tagged '${tag}'</h3>
#{/else}

*{********* Posts list *********}*

<div class="older-posts">
    #{list items:posts, as:'post'}
        #{display post:post, as:'teaser' /}
    #{/list}
</div>
```



Go to the next part.

Currently, we have no way to create new blog posts or moderate comments using the blog UI. Play provides an out of the box **CRUD** module that will help quickly generate a basic administration area.

Enabling the CRUD module

A Play application can be assembled from several application modules. This lets you reuse components across several applications or split a large application into several smaller ones.

The CRUD module is a generic application that introspects the model classes to create simple lists and forms.

To enable the CRUD module, add this line to the **/yabe/conf/application.conf** file:

```
# Import the crud module
module.crud=${play.path}/modules/crud
```

Then this module comes with a generic set of **routes** that we can reuse for now. To import these routes just add this line to the **/yabe/conf/routes** file:

```
# Import CRUD routes
*      /admin              module:crud
```

This will import all the CRUD routes using the **/admin** prefix for the URL paths.

You have to restart the application to take the new module into account.

Declaring the CRUD controllers

For each model object that we want to integrate to the administration area, we have to declare a controller that extends the **controllers.CRUD** super controller. This is very easy.

Create one controller for each model object. For example, for the **Post** object create the **Posts** controller in the **/yabe/app/controllers/Posts.java** file.

```
package controllers;

import play.*;
import play.mvc.*;

public class Posts extends CRUD {
}
```

The convention is to pluralize the model object name to create the corresponding controller. This way, Play will automatically find the associated model object for each controller. If you need to use a different name, you can still use the **@CRUD.For** annotation. Check the manual page.

Create the same for all model objects:

```
package controllers;

import play.*;
import play.mvc.*;

public class Users extends CRUD {
}
```

```
package controllers;

import play.*;
import play.mvc.*;

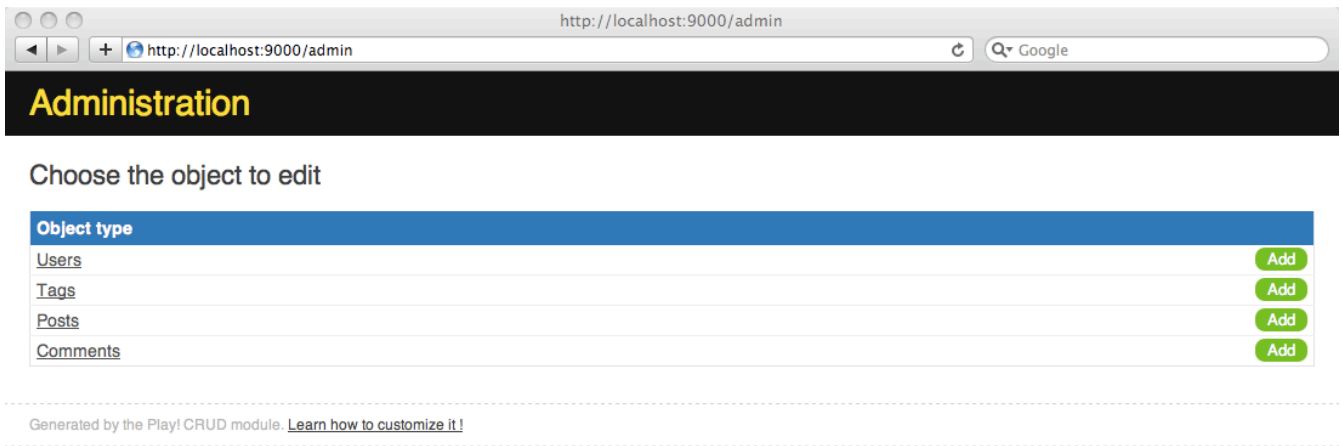
public class Comments extends CRUD {
}
```

```
package controllers;

import play.*;
import play.mvc.*;

public class Tags extends CRUD {
}
```

Now just open the http://localhost:9000/admin/ URL, and you should get to the administration area.



If you browse it a little, you will notice that the object names in the lists are a little rough. This is because the default is to use a simple **toString()** to get a readable representation of the model objects.

We can easily fix that, by providing correct implementations of the **toString()** method for all models. For example, for the User class:

```
...
public String toString() {
    return email;
}
...
```

Adding validation

The generated administration area's main problem is that forms don't contain any validation rules. But actually the CRUD module is able to extract the validation rules from the validation annotations if the model class is correctly annotated.

Let's add some annotations to the **User** class:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;
import play.data.validation.*;

@Entity
public class User extends Model {

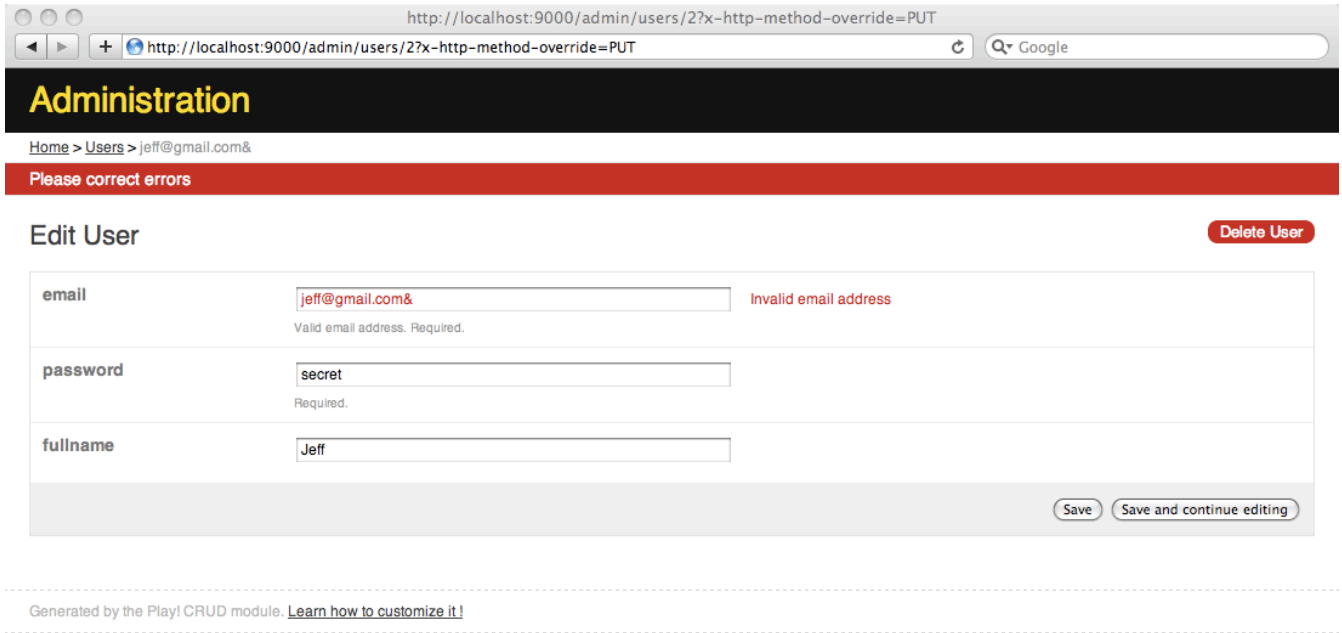
    @Email
    @Required
    public String email;

    @Required
    public String password;

    public String fullname;
    public boolean isAdmin;

...
```

Now if you go to the edition or creation form for the **User** model object, you will see that validation rules are auto-magically added to the form:



Let's do the same for the **Post** class:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;
import play.data.validation.*;

@Entity
public class Post extends Model {

    @Required
    public String title;

    @Required
    public Date postedAt;

    @Lob
    @Required
    @MaxSize(10000)
    public String content;

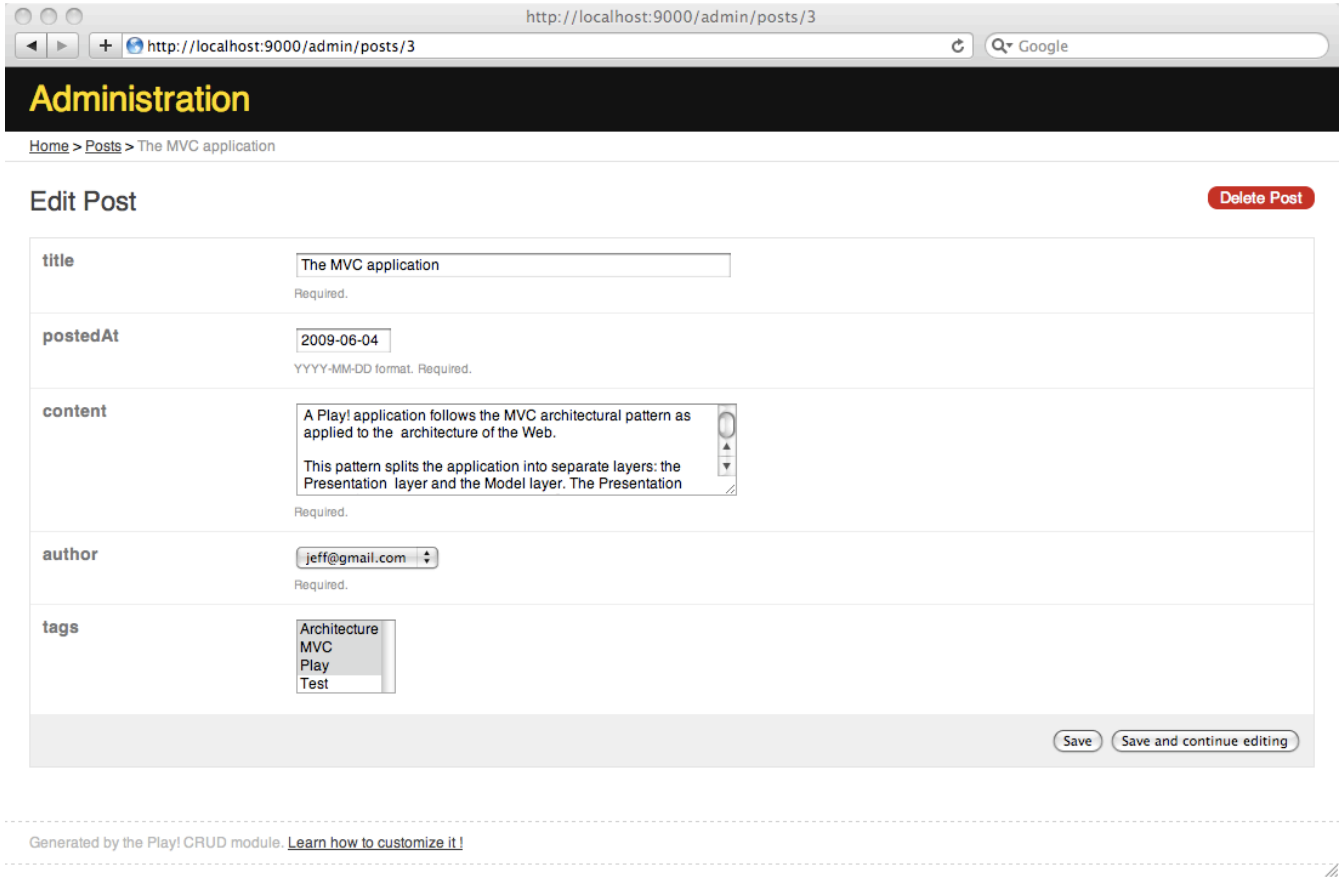
    @Required
    @ManyToOne
    public User author;

    @OneToMany(mappedBy="post", cascade=CascadeType.ALL)
    public List<Comment> comments;

    @ManyToMany(cascade=CascadeType.PERSIST)
    public Set<Tag> tags;

...
```

And check the result:



Here you see an interesting side effect: the **@MaxSize** validation rule has changed the way Play displays the Post form. It now uses a large text area for the content field.

Finally we can add validation rules to the **Comment** and **Tag** classes as well:

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;
import play.data.validation.*;

@Entity
public class Tag extends Model implements Comparable<Tag> {

    @Required
    public String name;

...
```

```
package models;

import java.util.*;
import javax.persistence.*;

import play.db.jpa.*;
import play.data.validation.*;

@Entity
public class Comment extends Model {

    @Required
    public String author;

    @Required
    public Date postedAt;

    @Lob
    @Required
    @MaxSize(10000)
    public String content;

    @ManyToOne
    @Required
    public Post post;

...
```

Better form labels

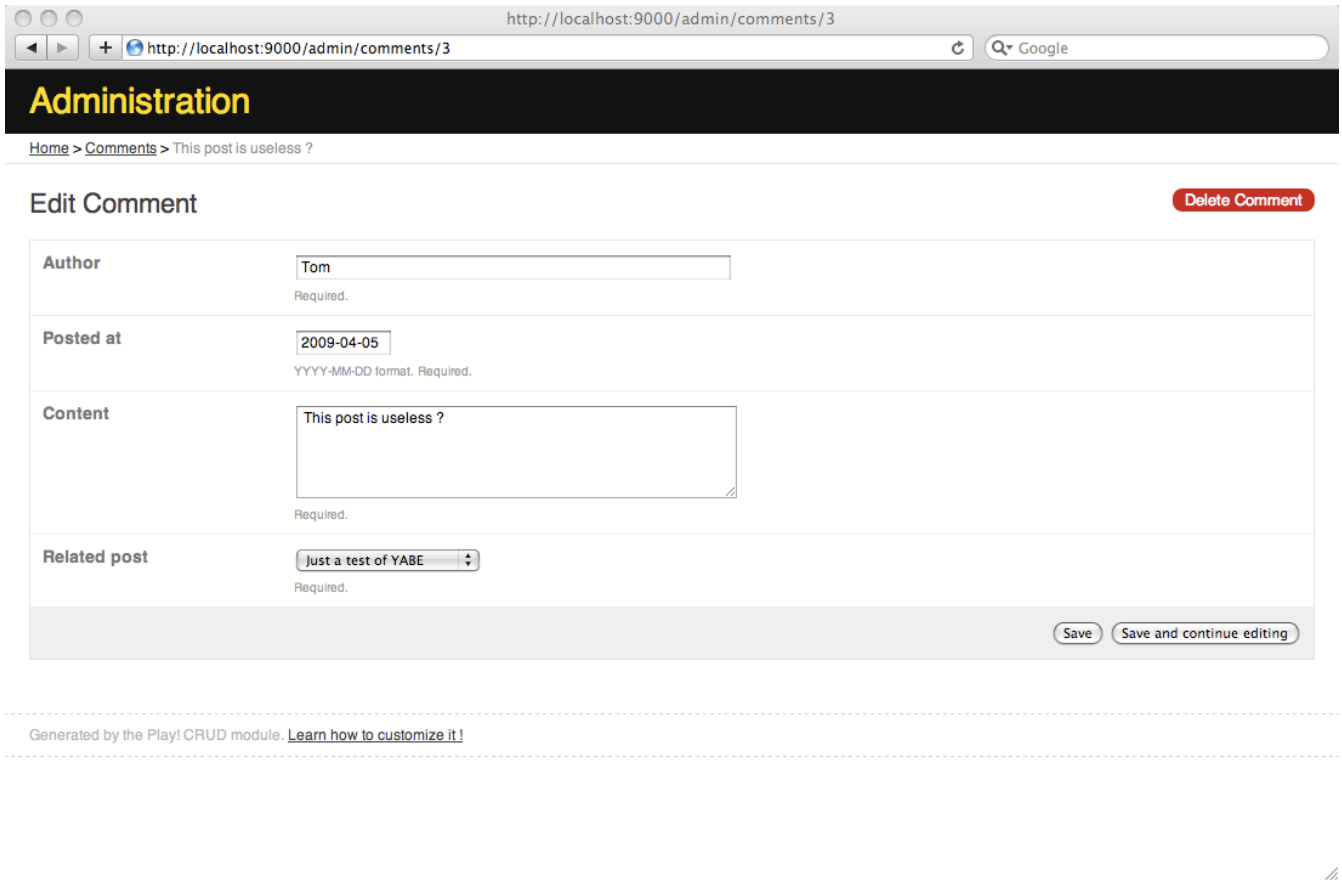
As you can see the form labels are a little rough. Play uses the Java field name as form label. To customize it, we just have to provide better labels in the **/yabe/conf/messages** file.

> In fact, you can have a separate **messages** file for each language supported by your application. For example, you could put French messages in the **/yabe/conf/messages.fr** file. You will see how to add language localisation in part 12: Internationalisation and localisation.

Add these labels to the **messages** file:

```
title=Title
content=Content
postedAt=Posted at
author=Author
post=Related post
tags=Tags set
name=Common name
email=Email
password=Password
fullname=Full name
isAdmin=User is admin
```

Then refresh any form, and you will see the new form labels:



Customizing the 'Comments' data list

The CRUD module is made to be fully customizable. For example if you look at the comments list page, the way that data is listed is not great. We would like to add more columns, especially the 'related post' column to help us filter the list easily.

In fact, as your applications keep the master, you can really override any **action** or **template** provided by the CRUD module. For example if we want to customize the 'comments list' view, we just have to provide another **/yabe/app/views/Comments/list.html** template.

The **CRUD** module provides more Play commands when it's enabled. The **crud:ov** command helps you override any template. From a command line, type:

```
$ play crud:ov --template Comments/list
```

Now you have a new template at **/yabe/app/views/Comments/list.html**:

```
#{extends 'CRUD/layout.html' /}

<div id="crudList" class="${type.name}">

        <h2 id="crudListTitle">&{'crud.list.title', type.name}</h2>

        <div id="crudListSearch">
                #{crud.search /}
        </div>

        <div id="crudListTable">
                #{crud.table /}
        </div>

        <div id="crudListPagination">
                #{crud.pagination /}
        </div>

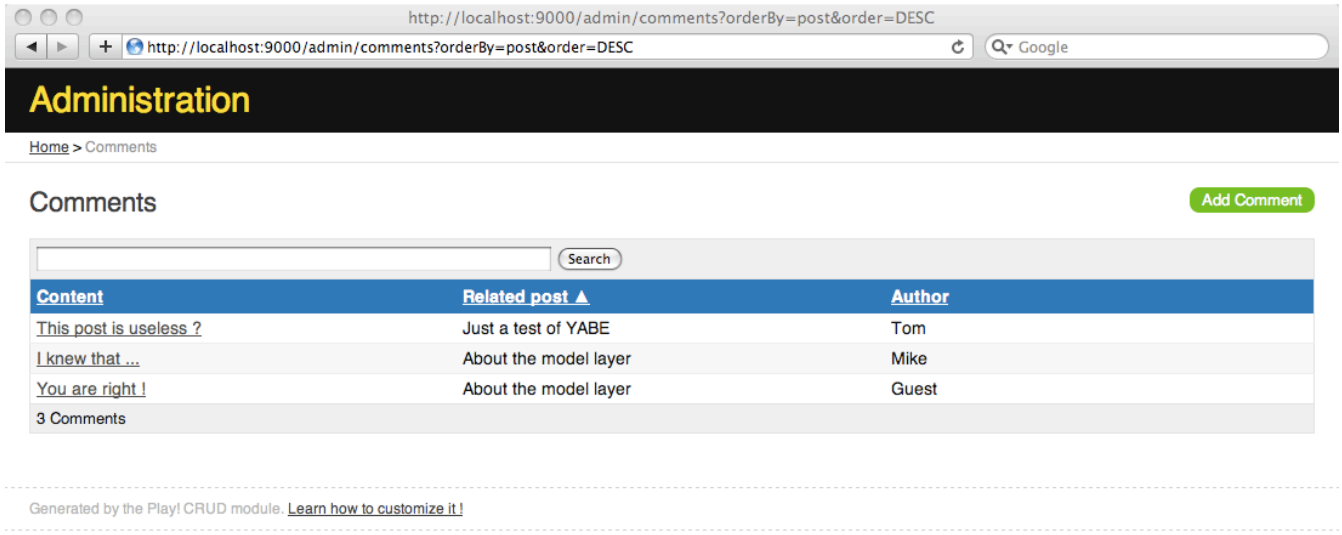
        <p id="crudListAdd">
                <a href="@{blank()}">&{'crud.add', type.modelName}</a>
        </p>

</div>
```

The **#{crud.table /}** actually generates the table. We can customize it using the **fields** parameter to add more columns. Try this:

```
#{crud.table fields:['content', 'post', 'author'] /}
```

And now we have 3 columns in the table:



The problem is that the **content** field could be very long for some comments. We will specialize the way that the **#{crud.table /}** handles it to be able to truncate it if needed.

We can specify a custom way to display each field using the **#{crud.custom /}** tag as is:

```
#{crud.table fields:['content', 'post', 'author']}
 #{crud.custom 'content'}
  <a href="@{Comments.show(object.id)}">
   ${object.content.length() > 50 ? object.content[0..50] + '…' :
object.content}
  </a>
 #{/crud.custom}
#{/crud.table}
```

Yes, there is some Groovy syntactic sugar at work here.

Customizing the 'Post' form

We can customize the generated forms as well. For example, the way we enter tags in the Post form is not really easy. We could build something better. Let's override the **Posts/show** template:

```
$ play crud:ov --template Posts/show
```

Now you have a new template at **/yabe/app/views/Posts/show.html**:

```
#{extends 'CRUD/layout.html' /}

<div id="crudShow" class="${type.name}">

<h2 id="crudShowTitle">&{'crud.show.title', type.modelName}</h2>

<div class="objectForm">
#{form action:@save(object.id), enctype:'multipart/form-data'}
    #{crud.form /}
    <p class="crudButtons">
        <input type="submit" name="_save" value="&{'crud.save',
type.modelName}" />
        <input type="submit" name="_saveAndContinue"
value="&{'crud.saveAndContinue', type.modelName}" />
    </p>
#{/form}
</div>

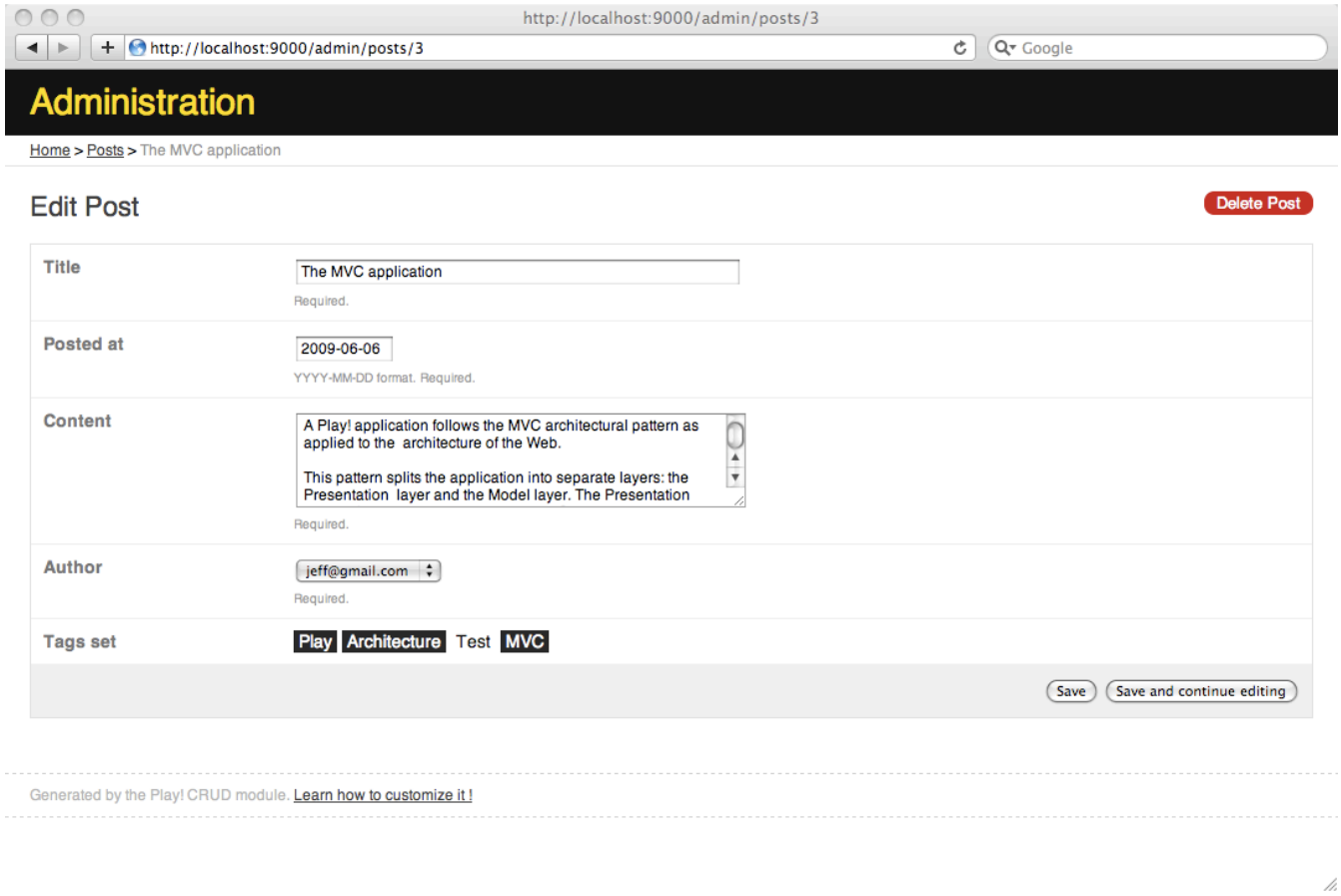
#{form @delete(object.id)}
    <p class="crudDelete">
        <input type="submit" value="&{'crud.delete', type.modelName}" />
    </p>
#{/form}

</div>
```

You can hack the **#{crud.form /}** tag the make the **tags** field custom:

```
#{crud.form}
    #{crud.custom 'tags'}
        <label for="tags">
            &{'tags'}
        </label>
        <style type="text/css">
                .tags-list .tag {
                    cursor: pointer;
                    padding: 1px 4px;
                }
                .tags-list .selected {
                    background: #222;
                    color: #fff;
                }
            </style>
            <script type="text/javascript">
                var toggle = function(tagEl) {
                    var input = document.getElementById('h'+tagEl.id);
                    if(tagEl.className.indexOf('selected') > -1) {
                        tagEl.className = 'tag';
                        input.value = '';
                    } else {
                        tagEl.className = 'tag selected';
                        input.value = tagEl.id;
                    }
                }
            </script>
            <div class="tags-list">
                #{list items:models.Tag.findAll(), as:'tag'}
                   <span id="${tag.id}" onclick="toggle(this)"
                        class="tag ${object.tags.contains(tag) ? 'selected' : ''}">
                       ${tag}
                   </span>
                   <input id="h${tag.id}" type="hidden" name="${fieldName}"
                            value="${object.tags.contains(tag) ? tag.id : ''}" />
                #{/list}
            </div>
    #{/crud.custom}
#{/crud.form}
```

This is a little hacky and we could do better here, but we have now a simpler tags selector using a little bit of JavaScript:



This is a good start for the administration area!

Go to the next part.

Now that we have an administration area we need to plug in some authentication. Fortunately, Play has a module for that. It's called the **Secure** module.

Enabling the Secure module

Enable the **SECURE** module in the **yabe/conf/application.conf** file and restart the application.

```
# Import the secure module
module.secure=${play.path}/modules/secure
```

After restarting, Play should say that the module is loaded.

The **Secure** module comes with a set of default routes that you can easily enable in the **yabe/conf/routes** file (or we could define our own routes as well):

```
# Import Secure routes
*       /                                       module:secure
```

Protecting the admin controllers

The module provides a **controllers.Secure** controller that declares all needed interceptors. Of course we could just inherit from this controller but because Java is limited to simple inheritance it would be problematic.

Instead of directly inheriting from the **Secure** controller, we can annotate the admin controllers using the **@With** annotation to tell Play to invoke the corresponding interceptors as well:

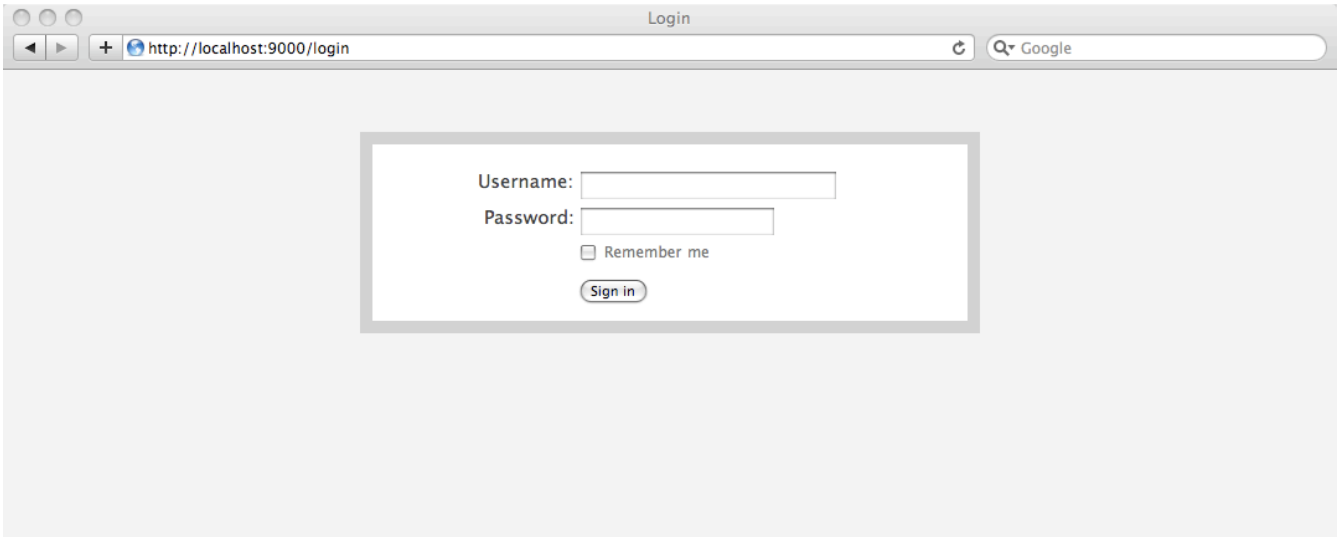
```
package controllers;

import play.*;
import play.mvc.*;

@With(Secure.class)
public class Posts extends CRUD {
}
```

Do the same for the **Comments**, **Users** and **Tags** controllers.

Now if you try to access any admin action, you should get a log-in page:



In fact, for now you can try any user name/password pair, and it will just work.

Customizing the authentication process

The application has to provide an instance of **controllers.Secure.Security** to customize the authentication process. By creating our own version of this class we will be able to specify exactly how users should be authenticated.

Create a **yabe/controllers/Security.java** file and override the **authenticate()** method:

```
package controllers;

import models.*;

public class Security extends Secure.Security {

    static boolean authenticate(String username, String password) {
        return true;
    }

}
```

As we already have the User objects part of the blog model, it is easy to implement a working version of this method:

```
static boolean authenticate(String username, String password) {
    return User.connect(username, password) != null;
}
```

Now go to http://localhost:9000/logout to log out and you can try logging in with one of the users imported in the **initial-data.yml** file, like **bob@gmail.com/secret**.

Refactoring the administration area

We've started the administration area using the CRUD module, but it is not yet well integrated with the blog UI. We will start to work on a new administration area. This one will give each author access to his/her own posts. The full-featured administration using CRUD will still be available for super administrator users.

Let's create a new **Admin** controller for the administration part:

```
package controllers;

import play.*;
import play.mvc.*;

import java.util.*;

import models.*;

@With(Secure.class)
public class Admin extends Controller {

    @Before
    static void setConnectedUser() {
        if(Security.isConnected()) {
            User user = User.find("byEmail", Security.connected()).first();
            renderArgs.put("user", user.fullname);
        }
    }

    public static void index() {
        render();
    }

}
```

And refactor the routes definition in **yabe/conf/routes**:

```
# Administration
GET     /admin/?                                Admin.index
*       /admin                                  module:crud
```

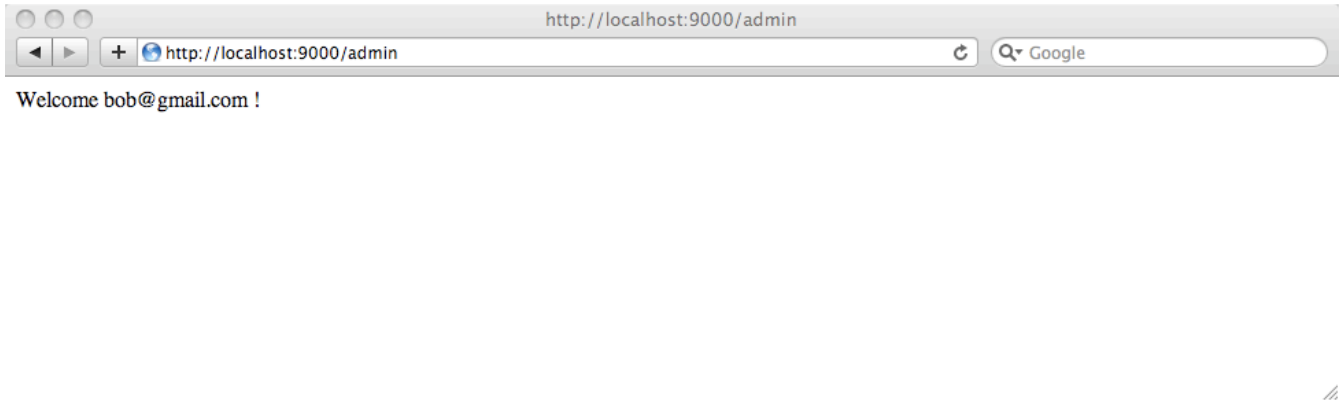
Now link the 'Log in to write something' text from the **yabe/app/views/main.html** template to this controller:

```
...
<ul id="tools">
    <li>
        <a href="@{Admin.index()}">Log in to write something</a>
    </li>
</ul>
...
```

The last thing to make all this new stuff work is to create the **yabe/app/views/Admin/index.html** template. Let's start with something simple:

```
Welcome ${user}!
```

Now, go to the blog home page, click the 'Log in to write something' link and you should get to the new administration area:



Good start! But since we will have several pages in this administration area, we need a super-template. Let's create it in the **yabe/app/views/admin.html** file:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>Administration</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        #{get 'moreStyles' /}
        <link rel="stylesheet" type="text/css" media="screen"
                href="@{'/public/stylesheets/main.css'}" />
        <link rel="shortcut icon" type="image/png"

href="@{'/public/file:///Users/nicolasleroux/Projects/Personal/1.1/documentation/images/
/>
        <script
src="@{'/public/javascripts/jquery-1.3.2.min.js'}"></script>
        <script
src="@{'/public/javascripts/jquery.tools.min.js'}"></script>
    </head>
    <body id="admin">

        <div id="header">
            <div id="logo">
                yabe. <span>administration</span>
            </div>
            <ul id="tools">
                <li>
                    <a href="@{Secure.logout()}">Log out</a>
                </li>
            </ul>
        </div>

        <div id="main">
            #{doLayout /}
        </div>

        <p id="footer">
            Yabe is a (not so) powerful blog engine built with the
            <a href="http://www.playframework.org">Play framework</a>
            as a tutorial application.
        </p>

    </body>
</html>
```

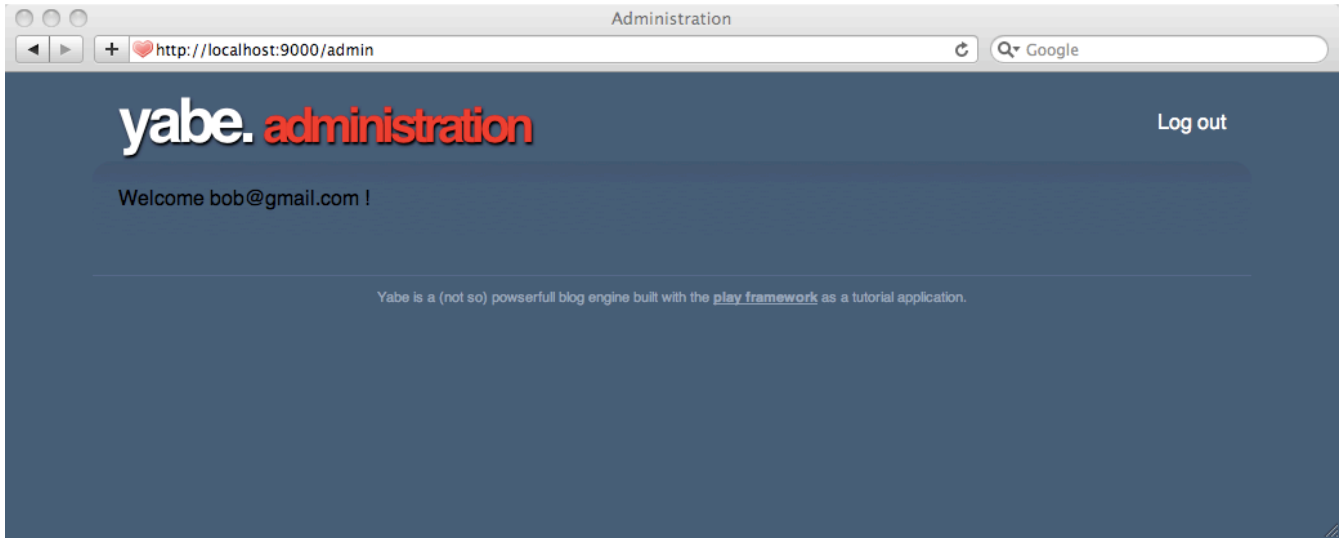
As you can see, it's very close to the template used for the front part of the blog engine. You have replaced the **Log in** link by a **Log out** link that calls the **Secure** controller's **logout** action provided by the secure module.

Now let's use it in the **yabe/app/views/Admin/index.html** template:

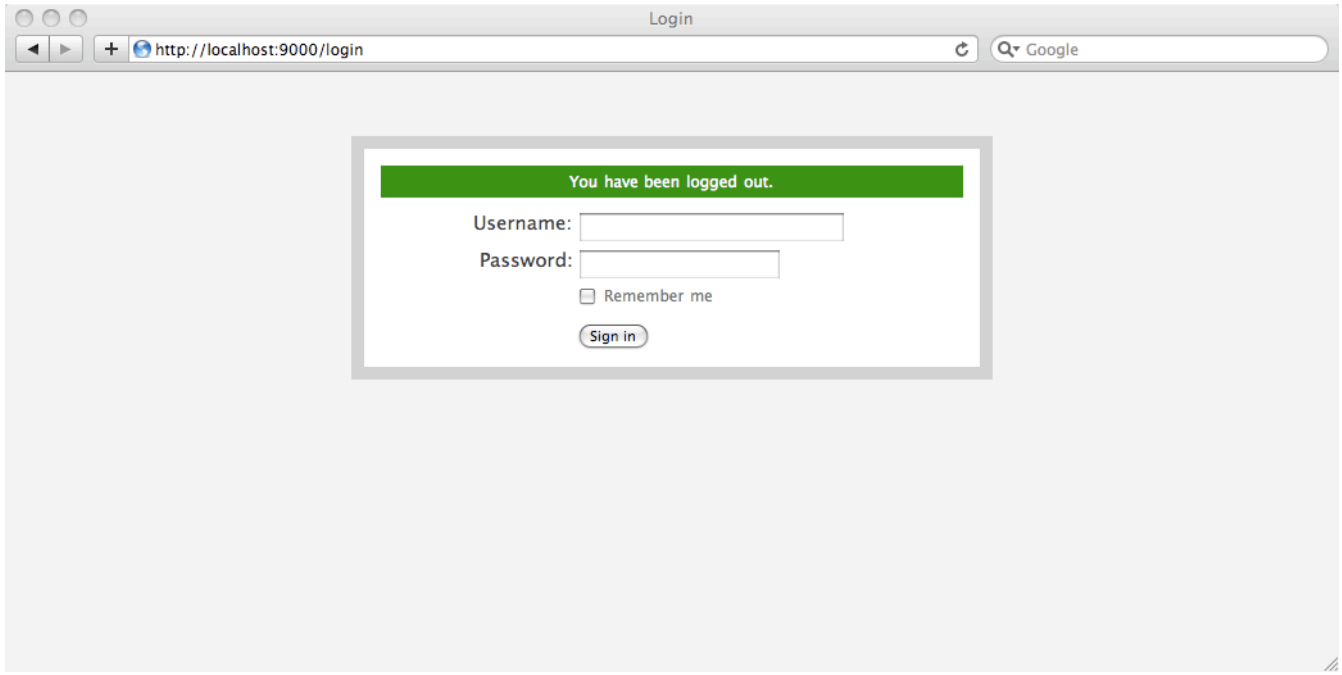
```
#{extends 'admin.html' /}

Welcome ${user}!
```

And refresh!



If you try the **log out** link it will send you back to the login form:



That's the default way for the secure module to handle log out events. But it's easily customizable simply by overriding the **onDisconnected()** method in the **controllers.Security** class:

```
static void onDisconnected() {
    Application.index();
}
```

You can do the same for the **onAuthenticated()** event:

```
static void onAuthenticated() {
    Admin.index();
}
```

Adding roles

Actually, we need two administration areas: one for simple editors and another one for the super administrator. As you've seen the **User** model object has an **isAdmin** field that indicates whether the user has super administrator rights.

The secure module not only provides **authentication** but **authorization** management too. It's called **profiles**. To create an **admin** profile, you just need to override the **check()** method in the **controllers.Security** class:

```
static boolean check(String profile) {
    if("admin".equals(profile)) {
        return User.find("byEmail", connected()).<User>first().isAdmin;
    }
    return false;
}
```

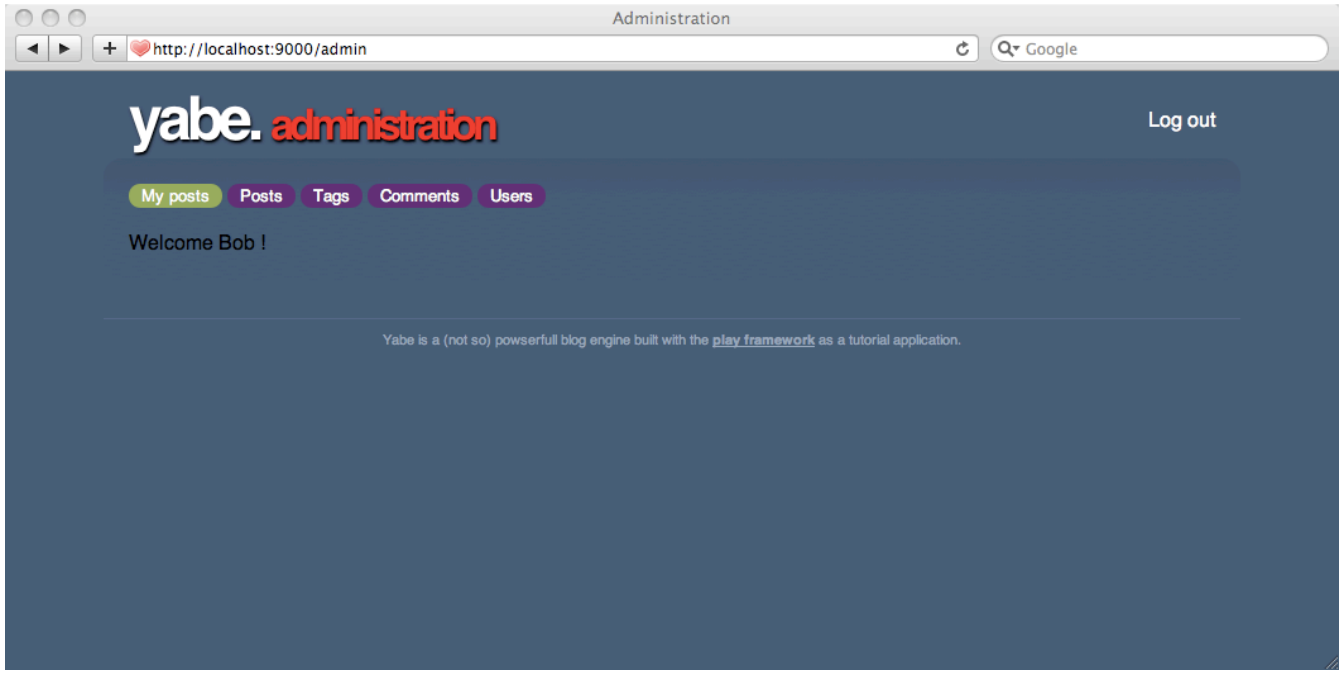
Now we can show an administration menu if the user has the admin role. Update the **yabe/views/admin.html** to integrate a top level menu:

```
...
<div id="main">

    <ul id="adminMenu">
        <li class="${request.controller == 'Admin' ? 'selected' : ''}">
            <a href="@{Admin.index()}">My posts</a>
        </li>
        #{secure.check 'admin'}
            <li class="${request.controller == 'Posts' ? 'selected' : ''}">
                <a href="@{Posts.list()}">Posts</a>
            </li>
            <li class="${request.controller == 'Tags' ? 'selected' : ''}">
                <a href="@{Tags.list()}">Tags</a>
            </li>
            <li class="${request.controller == 'Comments' ? 'selected' : ''}">
                <a href="@{Comments.list()}">Comments</a>
            </li>
            <li class="${request.controller == 'Users' ? 'selected' : ''}">
                <a href="@{Users.list()}">Users</a>
            </li>
        #{/secure.check}
    </ul>

    #{doLayout /}
</div>
...
```

Note how we used the **#{secure.check /}** tag to show part of the menu only to **admin** users.



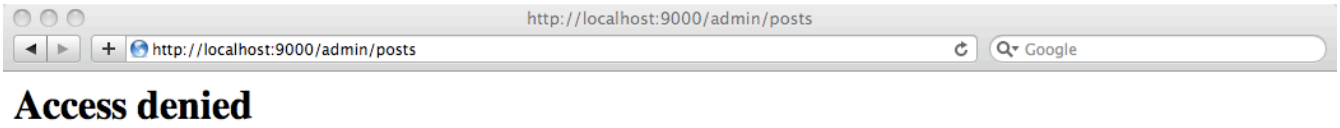
But that's not yet enough to secure the site's CRUD section! If a user knows the URL he can still access it. We have to protect the controllers as well. We can do so easily using the **@Check** annotation. For example, for the **Posts** controller:

```
package controllers;

import play.*;
import play.mvc.*;

@Check("admin")
@With(Secure.class)
public class Posts extends CRUD {
}
```

Do the same for the **Tags**, **Comments** and **Users** controllers. Now try to log out and log in again with a standard user (like **jeff@gmail.com/secret**). You should not see the CRUD admin links. And if you try to access the **http://localhost:9000/admin/users** URL directly you will get a 403 Forbidden response.



Customizing the CRUD layout

That's fun but when we go to the CRUD part of the administration area, we lose the administration layout. That's because the CRUD module provides it own layout. But we can of course override it. Use this Play command:

```
play crud:ov --layout
```

And you get a new **/yabe/app/views/CRUD/layout.html** template. Let's replace its content to integrate it nicely with our **admin.html** layout:

```
#{extends 'admin.html' /}
#{set 'moreStyles'}
    <link rel="stylesheet" type="text/css" media="screen"
        href="@{'/public/stylesheets/crud.css'}" />
#{/set}

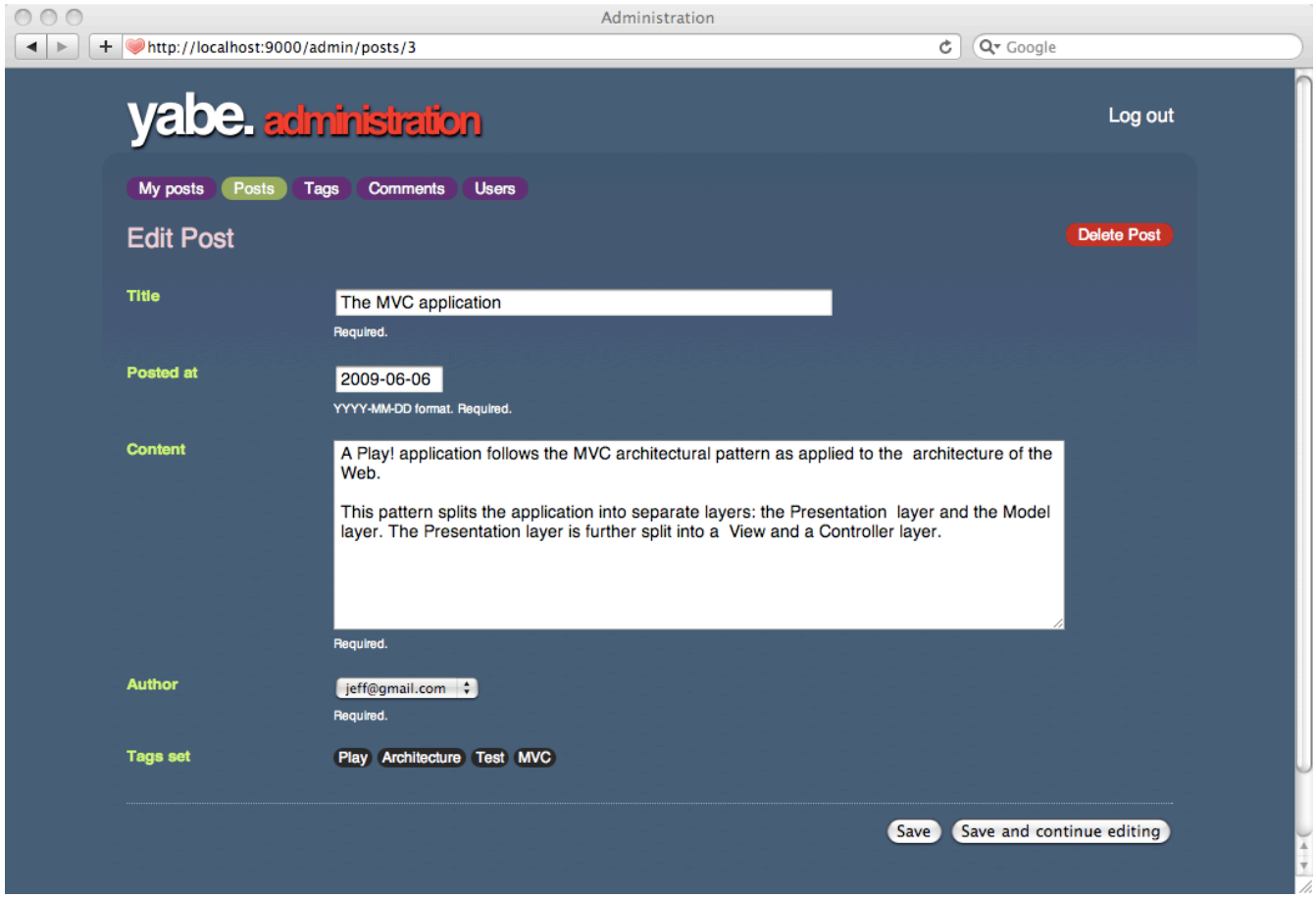
<div id="crud">

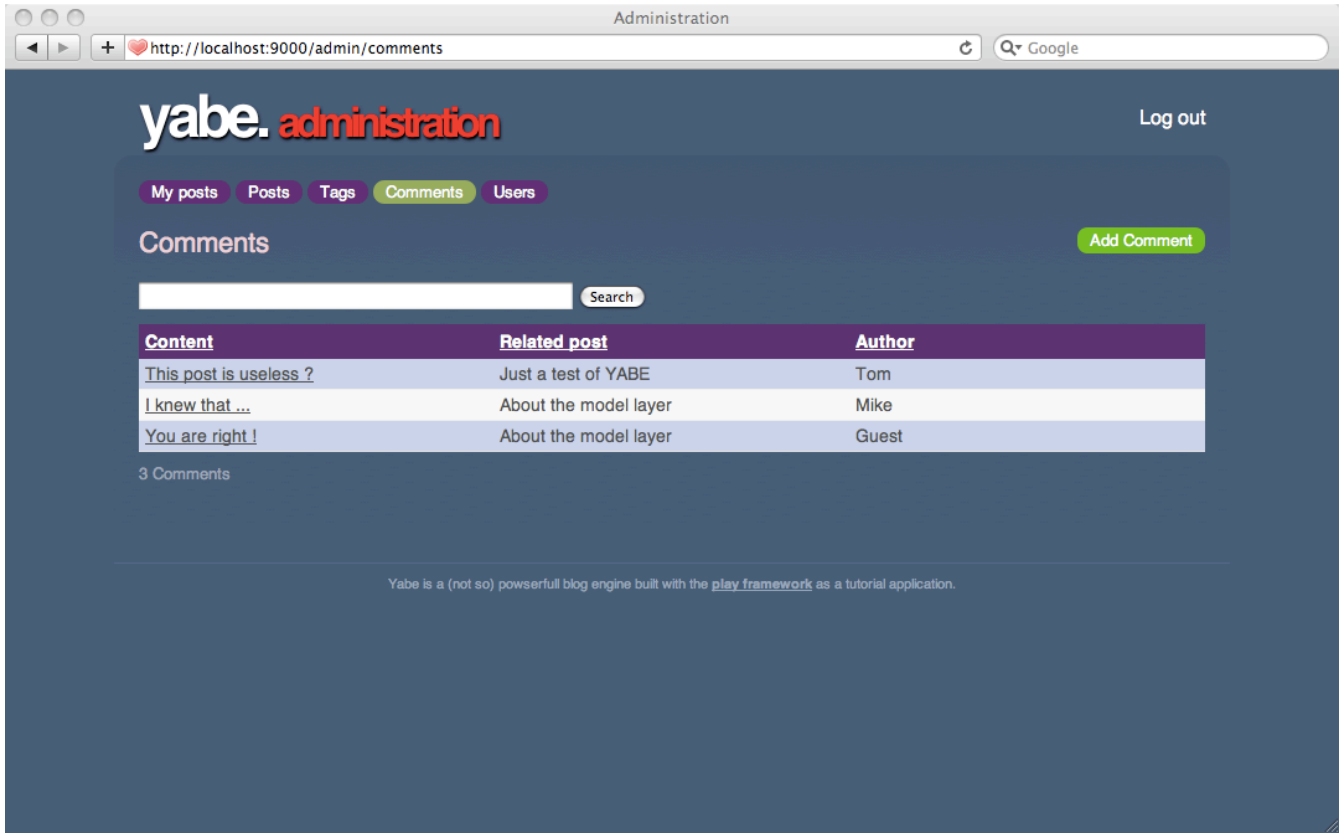
    #{if flash.success}
        <div class="crudFlash flashSuccess">
                ${flash.success}
        </div>
    #{/if}
    #{if flash.error || error}
        <div class="crudFlash flashError">
                ${error ?: flash.error}
        </div>
    #{/if}

    <div id="crudContent">
        #{doLayout /}
    </div>

</div>
```

As you can see, we reuse **crud.css** and include it in **admin.html** using the **get/set** template variables mechanism. Now if you try the CRUD part of the administration area, it's correctly integrated with the administration layout:


Administration
http://localhost:9000/admin/posts/3
Google
yabe. administration
Log out
My posts
Posts
Tags
Comments
Users
Edit Post
Delete Post
Title
The MVC application
Required.
Posted at
2009-06-06
YYYY-MM-DD format. Required.
Content
A Play! application follows the MVC architectural pattern as applied to the architecture of the Web.
This pattern splits the application into separate layers: the Presentation layer and the Model layer. The Presentation layer is further split into a View and a Controller layer.
Required.
Author
jeff@gmail.com
Required.
Tags set
Play
Architecture
Test
MVC
Save
Save and continue editing



Styling the login page

The administration area is almost graphically coherent now. The last thing to style is the log-in page. As always it is easily customizable by overriding the default CSS:

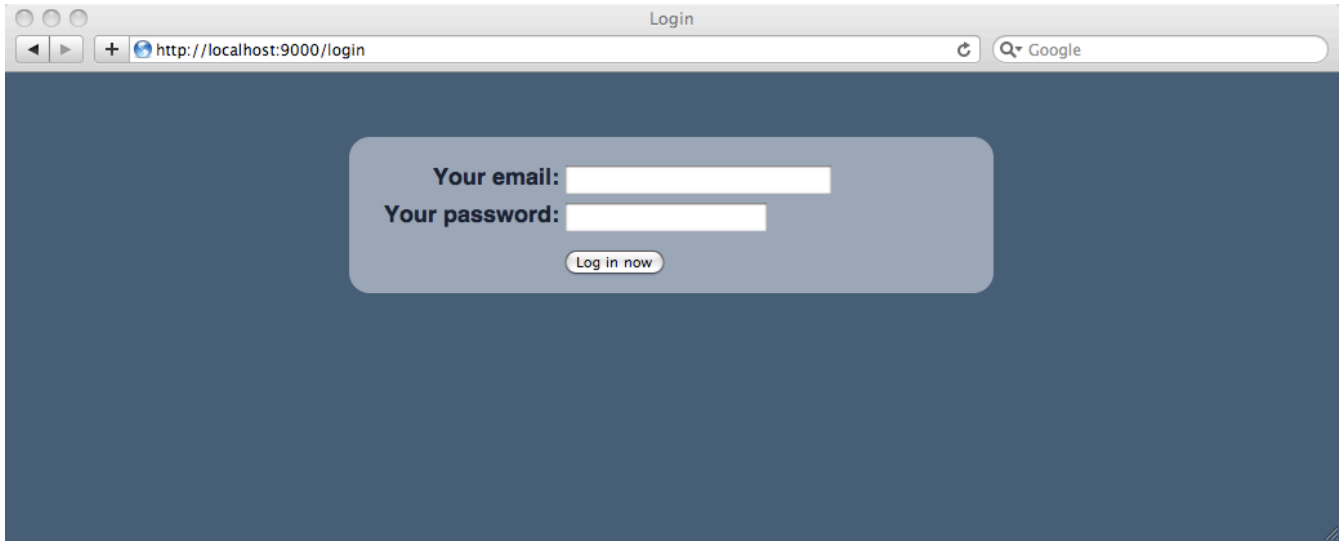
```
play secure:ov --css
```

We will just keep the CSS as is, but import our **main.css** at the top. Just add this first line to the **yabe/public/stylesheets/secure.css**:

```
@import url(main.css);
...
```

And customize the log-in screen messages, by adding these lines to your own **yabe/conf/messages** file:

```
secure.username=Your email:
secure.password=Your password:
secure.signin=Log in now
```



Go to the next part.

In the previous part we have created an administration area for yabe and prepared the 'My posts' section. This page will give each author a list of his own posts, and the ability to edit them or create a new one as well.

We could easily reuse the CRUD module as a basis for this page but here we will create something from scratch. We need a lot of personalization on these screens.

Start with the user posts list

We just need to retrieve and display the posts written by the connected user. Very easy. Start to enhance the **Admin.index** action:

```
public static void index() {
    List<Post> posts = Post.find("author.email", Security.connected()).fetch();
    render(posts);
}
```

And complete the **yabe/app/views/Admin/index.html**:

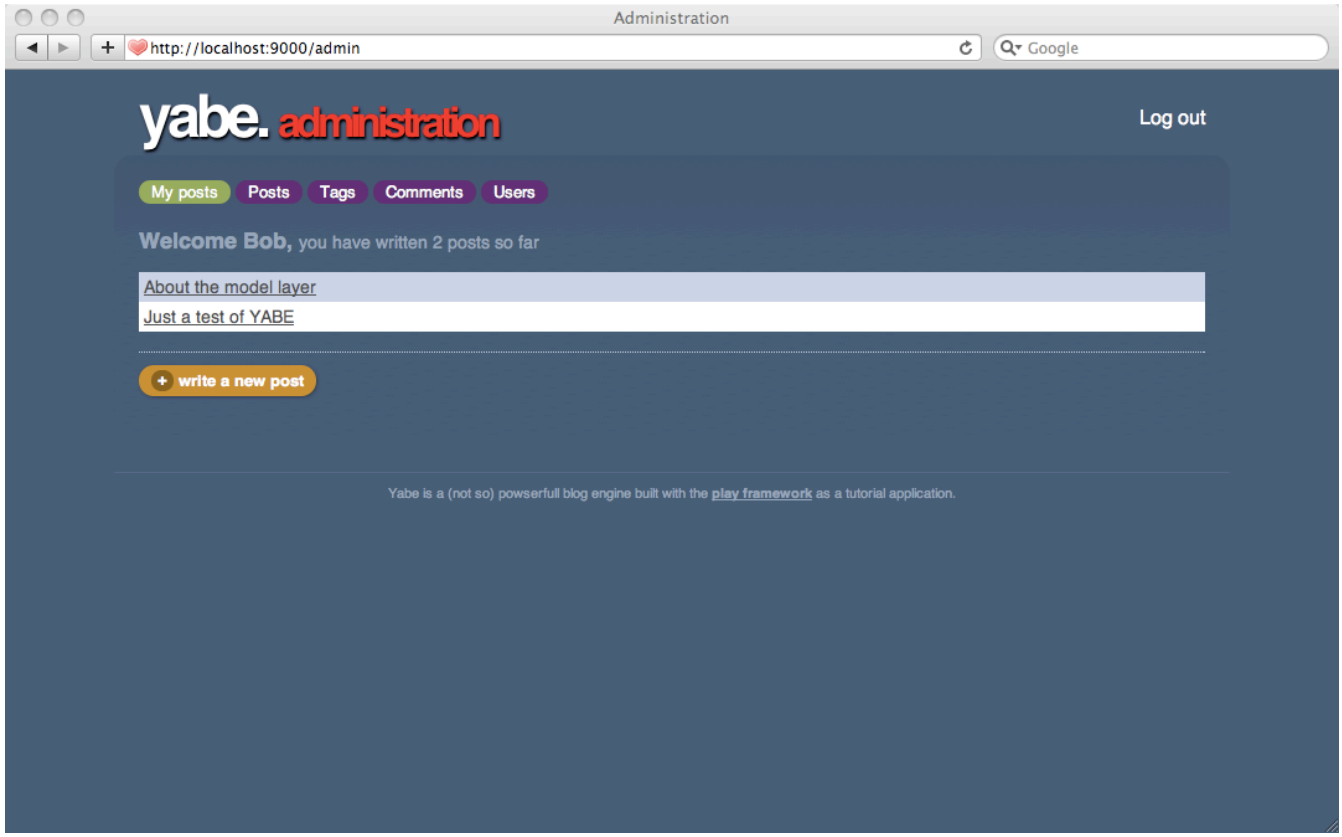
```
#{extends 'admin.html' /}

<h3>Welcome ${user}, <span>you have written ${posts.size() ?: 'no'}
${posts.pluralize('post', 'posts')} so far</span></h3>

#{list items:posts, as:'post'}
    <p class="post ${post_parity}">
        <a href="#">${post.title}</a>
    </p>
#{/list}

<p id="newPost" >
    <a href="#"><span>+</span> write a new post</a>
</p>
```

The first screen is ready:



The 'write a new post' page

We will create a form to create a new post. Basically with a form you have 2 actions: one to display the form and one to handle the form submission. Let's create the new **Admin.form** and **Admin.save** actions that will display and handle form submission:

Add the routes in the **yabe/conf/routes** file:

```
GET     /admin/new                          Admin.form
POST    /admin/new                          Admin.save
```

And add the **form()** and **save()** actions to the **Admin.java** controller:

```
public static void form() {
    render();
}

public static void save() {
    // Not implemented yet
}
```

Now you have to create the **yabe/app/views/Admin/form.html** template:

```
#{extends 'admin.html' /}

<h3>Write, <span>a new post</span></h3>

#{form @save()}

    #{ifErrors}
        <p class="error">
            Please correct these errors.
        </p>
    #{/ifErrors}

    <p>
        #{field 'title'}
        <label>Post title:</label>
        <input type="text" name="${field.name}"
            value="${post?.title}" class="${field.errorClass}" />
        #{/field}
    </p>

    <p>
        #{field 'content'}
        <label>Write here:</label>
        <textarea name="${field.name}"
            class="${field.errorClass}">${post?.content}</textarea>
        #{/field}
    </p>

    <p>
        #{field 'tags'}
        <label>Enter some tags:</label>
        <input type="text" size="50"
            name="${field.name}" value="${post?.tags?.join(' ')}" />
        #{/field}
    </p>

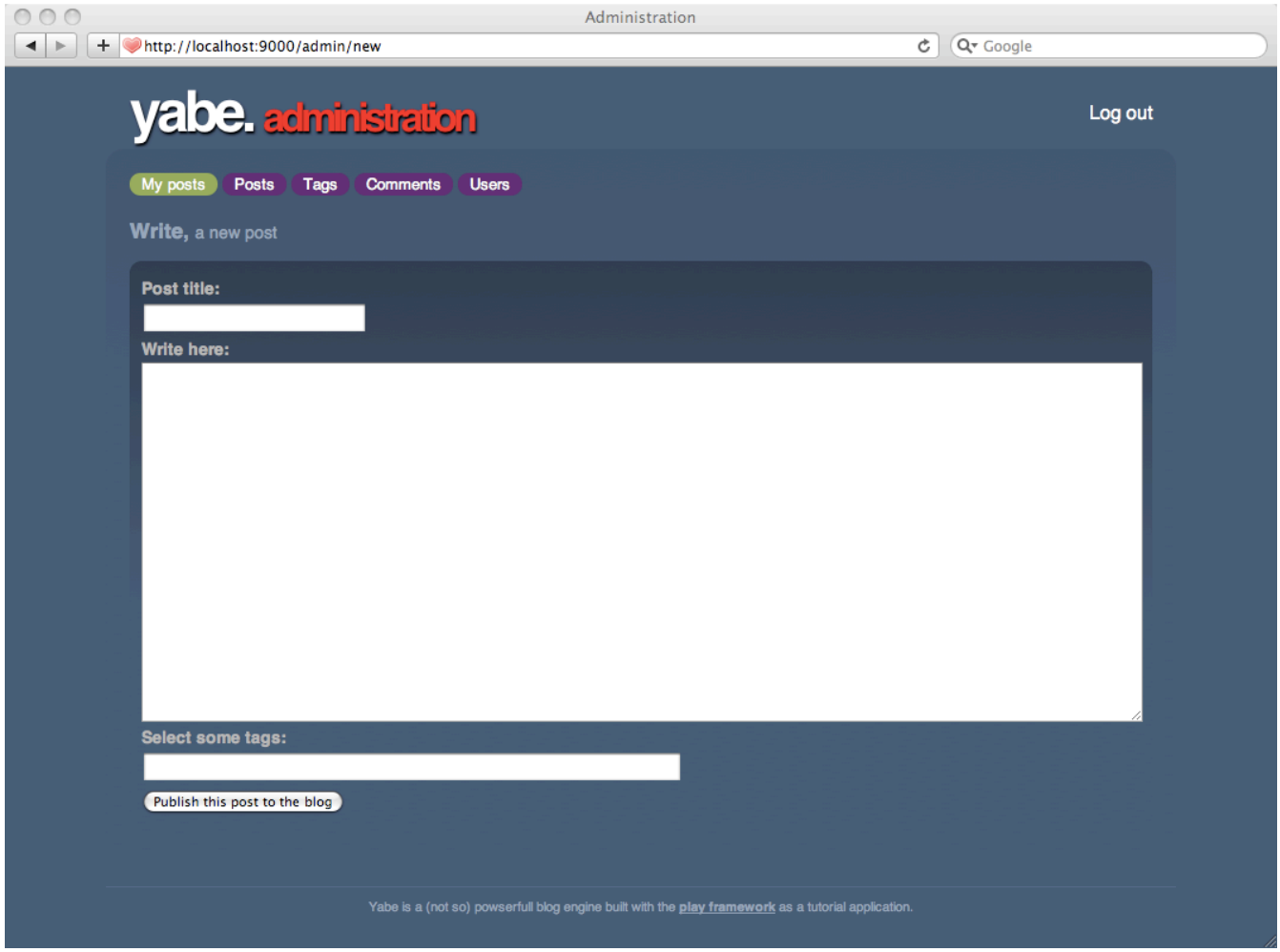
    <p>
        <input type="submit" value="Publish this post to the blog" />
    </p>

#{/form}
```

Finally, edit the **yabe/app/views/Admin/index.html** template to link the **Write a new post** button to this form:

```
...
<p id="newPost" >
    <a href="@{form()}"><span>+</span> write a new post</a>
</p>
...
```

You can check the result:



Now we have to complete the **Admin.save** action to handle correctly the form submission. It will create a new **Post** object, convert the tag list to a real **Tag** objects set, validate all the fields and save it. If there is any problem it will redisplay the form to show the errors.

```
public static void save(String title, String content, String tags) {
    // Create post
    User author = User.find("byEmail", Security.connected()).first();
    Post post = new Post(author, title, content);
    // Set tags list
    for(String tag : tags.split("\\s+")) {
        if(tag.trim().length() > 0) {
            post.tags.add(Tag.findOrCreateByName(tag));
        }
    }
    // Validate
    validation.valid(post);
    if(validation.hasErrors()) {
        render("@form", post);
    }
    // Save
    post.save();
    index();
}
```

Here we use render("@form") as a shortcut of render("Admin/form.html"). It just tells Play to use the form action's default template.

Test it!

Reusing this stuff for editing Posts

We have defined the HTML form and the Java action able to create a new blog Post. But we also need to allow editing existing posts. We can easily reuse exactly the same code with a few modifications.

First we need the **Admin.form** to retrieve an existing Post:

```
public static void form(Long id) {
    if(id != null) {
        Post post = Post.findById(id);
        render(post);
    }
    render();
}
```

As you see we made it optional, so the same action will retrieve an existing post only if the **id** parameter is filled. So you can now link the main screen's post list to the edit form. Edit the **yabe/app/views/Admin/index.html** template:

```
#{extends 'admin.html' /}

<h3>Welcome ${user}, <span>you have written ${posts.size() ?: 'no'}
${posts.pluralize('post', 'posts')} so far</span></h3>

#{list items:posts, as:'post'}
    <p class="post ${post_parity}">
        <a href="@{Admin.form(post.id)}">${post.title}</a>
    </p>
#{/list}

<p id="newPost" >
        <a href="@{form()}"><span>+</span> write a new
post</a>
</p>
```

Well, very easy but we have a problem. If you look at the actual URL generated by the Router for these links, you should see something like:

```
/admin/new?id=3
```

It works, but it's not really pretty. We will define another route that will use a different URL pattern if the **id** parameter is submitted:

```
GET     /admin/myPosts/{id}                 Admin.form
GET     /admin/new                          Admin.form
```

As you can see we define this route before the old one, so it has a higher priority. That means that if an **id** parameter is submitted Play will prefer this URL pattern. If not, it will still use the old route.

Refresh the **My posts** page and you should get better URL on these links.

Now we need to modify the **yabe/app/views/Admin/form.html** template as well:

```
#{extends 'admin.html' /}

#{ifnot post?.id}
    <h3>Write, <span>a new post</span></h3>
#{/ifnot}
#{else}
    <h3>Edit, <span>this post</span></h3>
#{/else}

#{form @save(post?.id)}

    #{ifErrors}
        <p class="error">
            Please correct these errors.
        </p>
    #{/ifErrors}

    <p>
        #{field 'title'}
        <label>Post title:</label>
        <input type="text" name="${field.name}"
            value="${post?.title}" class="${field.errorClass}" />
        #{/field}
    </p>

    <p>
        #{field 'content'}
        <label>Write here:</label>
        <textarea name="${field.name}"
            class="${field.errorClass}">${post?.content}</textarea>
        #{/field}
    </p>

    <p>
        #{field 'tags'}
        <label>Enter some tags:</label>
        <input type="text" size="50"
            name="${field.name}" value="${post?.tags?.join(' ')}" />
        #{/field}
    </p>

    <p>
        <input type="submit" value="Publish this post to the blog" />
    </p>

#{/form}
```

As you can see we have updated the form destination action to add the post ID as the first action parameter if it exists. So when the post has an id field set (meaning that the post already exists in the system), it will be sent the the **Admin.save** action.

Now we can change the **save()** method a little to handle both creation and editing cases:

```
public static void save(Long id, String title, String content, String tags) {
    Post post;
    if(id == null) {
        // Create post
        User author = User.find("byEmail", Security.connected()).first();
        post = new Post(author, title, content);
    } else {
        // Retrieve post
        post = Post.findById(id);
        // Edit
        post.title = title;
        post.content = content;
        post.tags.clear();
    }
    // Set tags list
    for(String tag : tags.split("\\s+")) {
        if(tag.trim().length() > 0) {
            post.tags.add(Tag.findOrCreateByName(tag));
        }
    }
    // Validate
    validation.valid(post);
    if(validation.hasErrors()) {
        render("@form", post);
    }
    // Save
    post.save();
    index();
}
```

And for better URL we will use the same trick as before, by adding a new preferred route if an **id** parameter exist:

```
POST    /admin/myPosts/{id}                 Admin.save
POST    /admin/new                          Admin.save
```

That's done! Now we're using the same action for both creating new blog posts and editing old ones, and the administration area is finished!

Go to the next part.

We've now finished the blog engine we wanted to create in this tutorial. However the project itself is not yet completely finished. To be totally confident with our code we need to add more tests to the project.

Of course we've already written unit tests in order to test all the yabe model layer functionality. And it's great as it will ensure that the blog engine's core functionality is well tested. But a web application is not only about the 'model' part. We need to ensure that the web interface works as expected. That means testing the yabe blog engine's controller layer. But we even need to test the UI itself, as for example, our JavaScript code.

Testing the controller part

Play gives you a way to test directly the application's controller part using JUnit. We call these tests **'Functional tests'**. This is because we want to test the web application's complete functionality.

Basically a functional test calls the Play **ActionInvoker** directly, simulating an HTTP request. So we give an HTTP method, an URI and HTTP parameters. Play then routes the request, invokes the corresponding action and sends you back the filled response. You can then analyze it to check that the response content is like you expected.

Let's write a first functional test. Open the **yabe/test/ApplicationTest.java** unit test:

```
import org.junit.*;
import play.test.*;
import play.mvc.*;
import play.mvc.Http.*;
import models.*;

public class ApplicationTest extends FunctionalTest {

    @Test
    public void testThatIndexPageWorks() {
        Response response = GET("/");
        assertIsOk(response);
        assertContentType("text/html", response);
        assertCharset("utf-8", response);
    }

}
```

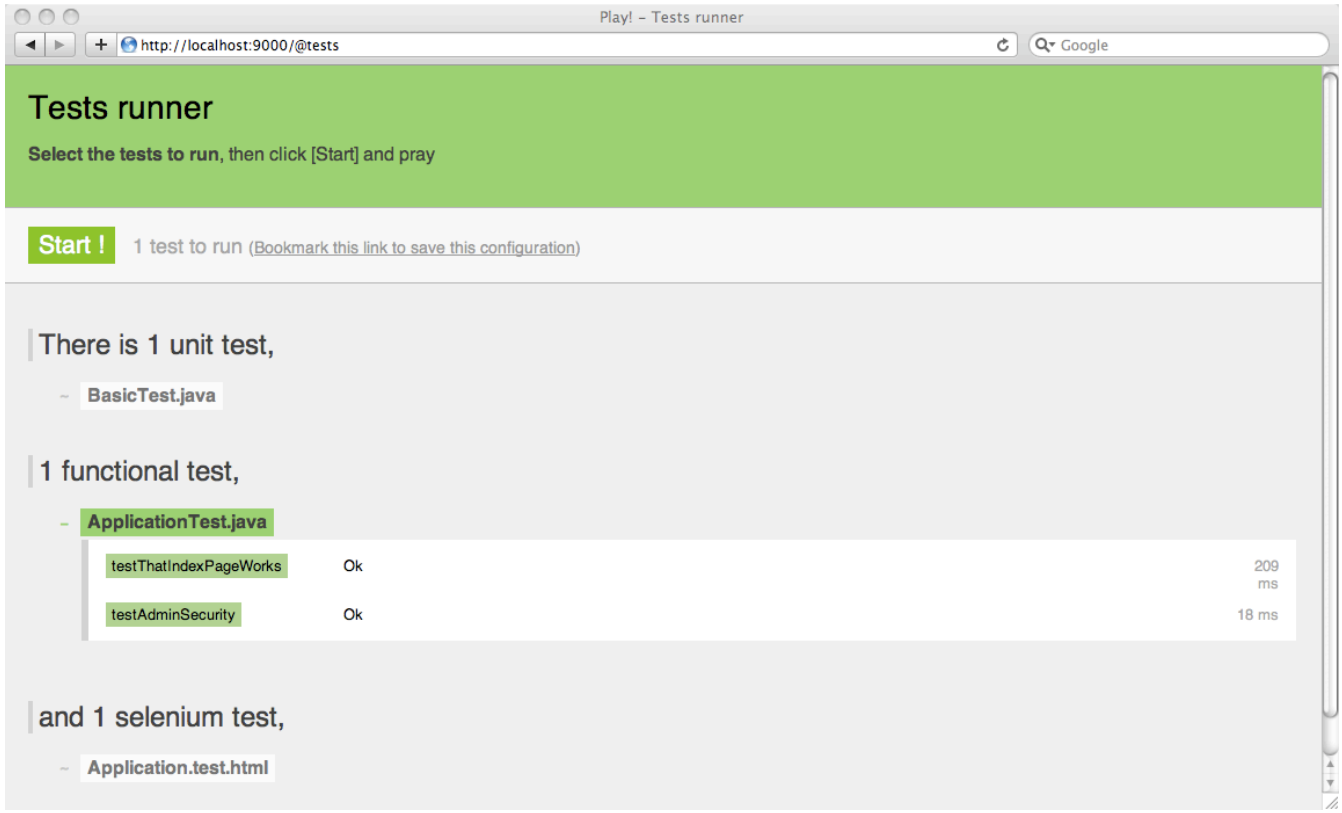
It looks like a standard JUnit test for now. Note that we use the Play **FunctionalTest** super class in order to get all useful utility helpers. This test is correct and just checks that the application home page (typically the '/' URL renders an HTML response with **'200 OK'** as status code).

Now we will check that the administration area's security works as expected. Add this new test to the **ApplicationTest.java** file:

```
...
@Test
public void testAdminSecurity() {
    Response response = GET("/admin");
    assertStatus(302, response);
    assertHeaderEquals("Location", "http://localhost/login", response);
}
...
```

Now run the yabe application in test mode using the **'play test'** command, open http://localhost:9000/@tests, select the **ApplicationTest.java** test case and run it.

Is it green?



Well, we could continue to test all the application functionalities this way, but it's not the best way to test an HTML-based web application. As our blog engine is intended to be executed in a web browser, it would be better to test it directly in a **real web browser**. And that's exactly what Play's **'Selenium tests'** do.

These kinds of JUnit based **'Functional tests'** are still useful, typically to test Web services returning non-HTML responses such as JSON or XML over HTTP.

Writing Selenium tests

Selenium is a testing tool specifically for testing web applications. The cool things here is that Selenium allows to run the test suite directly in any existing browser. As it does not use any 'browser simulator', you can be sure that you're testing what your users will use.

A Selenium test suite is typically written as an HTML file. The HTML syntax required by Selenium is a little tedious to write (formatted using an HTML table element). The good news is that Play helps you generate it using the Play template engine and a set of tags that support a simplified syntax for Selenium scenarios. An interesting side effect of using templates is that your are not tied to 'static scenarios' any more and you can use the power of Play templates (looping, conditional blocks) to write more complicated tests.

> However you can still write plain HTML Selenium syntax in the template and forget the specific Selenium tags if needed. It can become interesting if you use one of the several Selenium tools that generates the test scenarios for you, like Selenium IDE.

The default test suite of a newly-created Play application already contains a Selenium test. Open the **yabe/test/Application.test.html** file:

```
*{ You can use plain Selenium commands using the selenium tag }*

#{selenium}
    // Open the home page, and check that no error occurred
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
#{/selenium}
```

This test should run without any problem with the yabe application. It just opens the home page and checks that the page content does not contain the 'Application error' text.

However like any complex test, you need to set-up a set of well-known data before navigating the application and testing it. We will of course reuse the fixture concept and the **yabe/test/data.yml** file that we've used before. To import this data set before the test suite, just use the **#{fixtures /}** tag:

```
#{fixture delete:'all', load:'data.yml' /}

#{selenium}
    // Open the home page, and check that no error occurred
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
#{/selenium}
```

Another important thing to check is that we have a fresh user session at the test start. The session being stored in a browser transient cookie, you would keep the same session during two successive test runs.

So let's start our test with a special command:

```
#{fixture delete:'all', load:'data.yml' /}

#{selenium}
    clearSession()

    // Open the home page, and check that no error occurred
    open('/')
    waitForPageToLoad(1000)
    assertNotTitle('Application error')
#{/selenium}
```

Run it to be sure that there is no mistake. It should be green.

So we can write a more specific test. Open the home page and check that the default posts are present:

```
#{fixture delete:'all', load:'data.yml' /}

#{selenium 'Check home page'}
    clearSession()

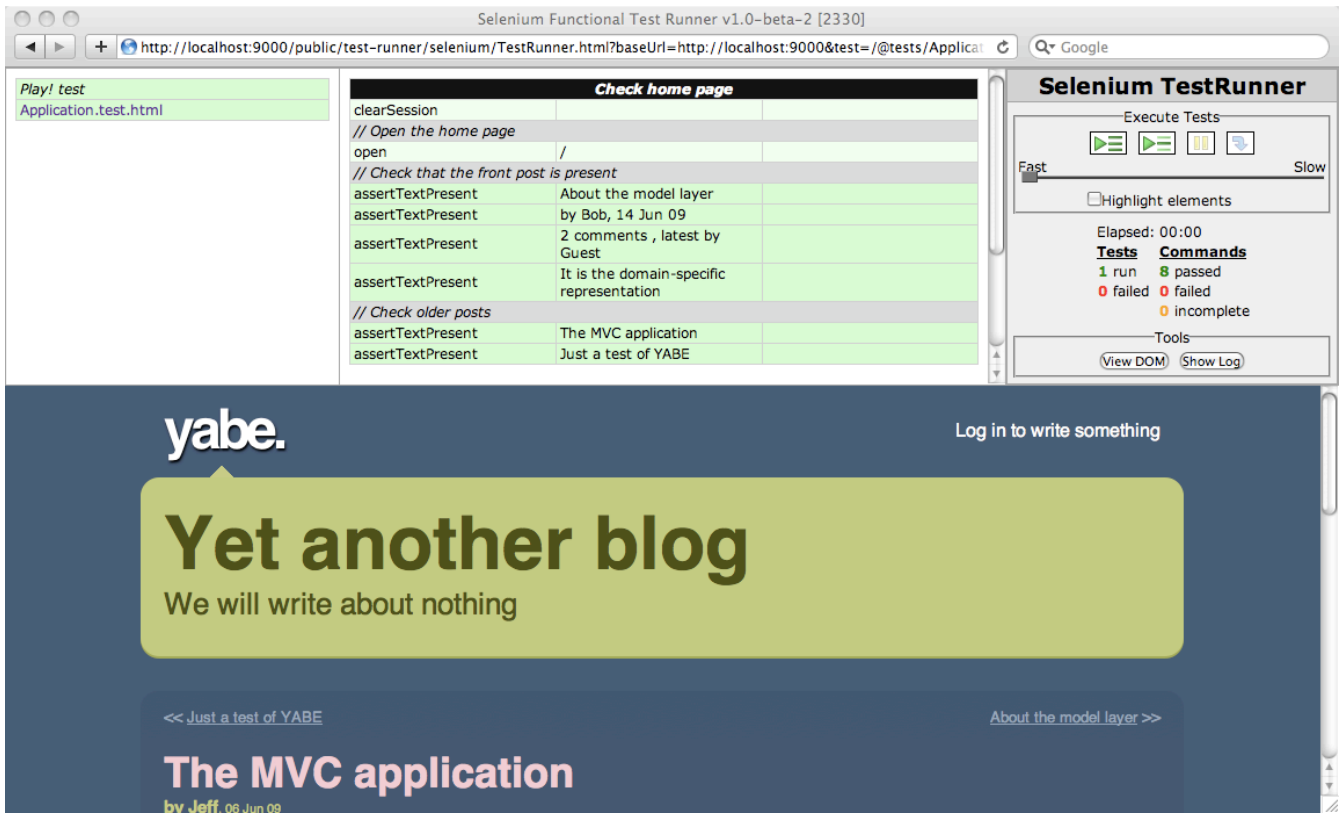
    // Open the home page
    open('/')

    // Check that the front post is present
    assertTextPresent('About the model layer')
    assertTextPresent('by Bob, 14 Jun 09')
    assertTextPresent('2 comments , latest by Guest')
    assertTextPresent('It is the domain-specific representation')

    // Check older posts
    assertTextPresent('The MVC application')
    assertTextPresent('Just a test of YABE')
#{/selenium}
```

We use the standard Selenium syntax, called selenese.

Run it (you can run in a different window just by opening the test link in a new window).



We will now test the comments form. Just add a new **#{selenium /}** tag to the template:

```
#{selenium 'Test comments'}

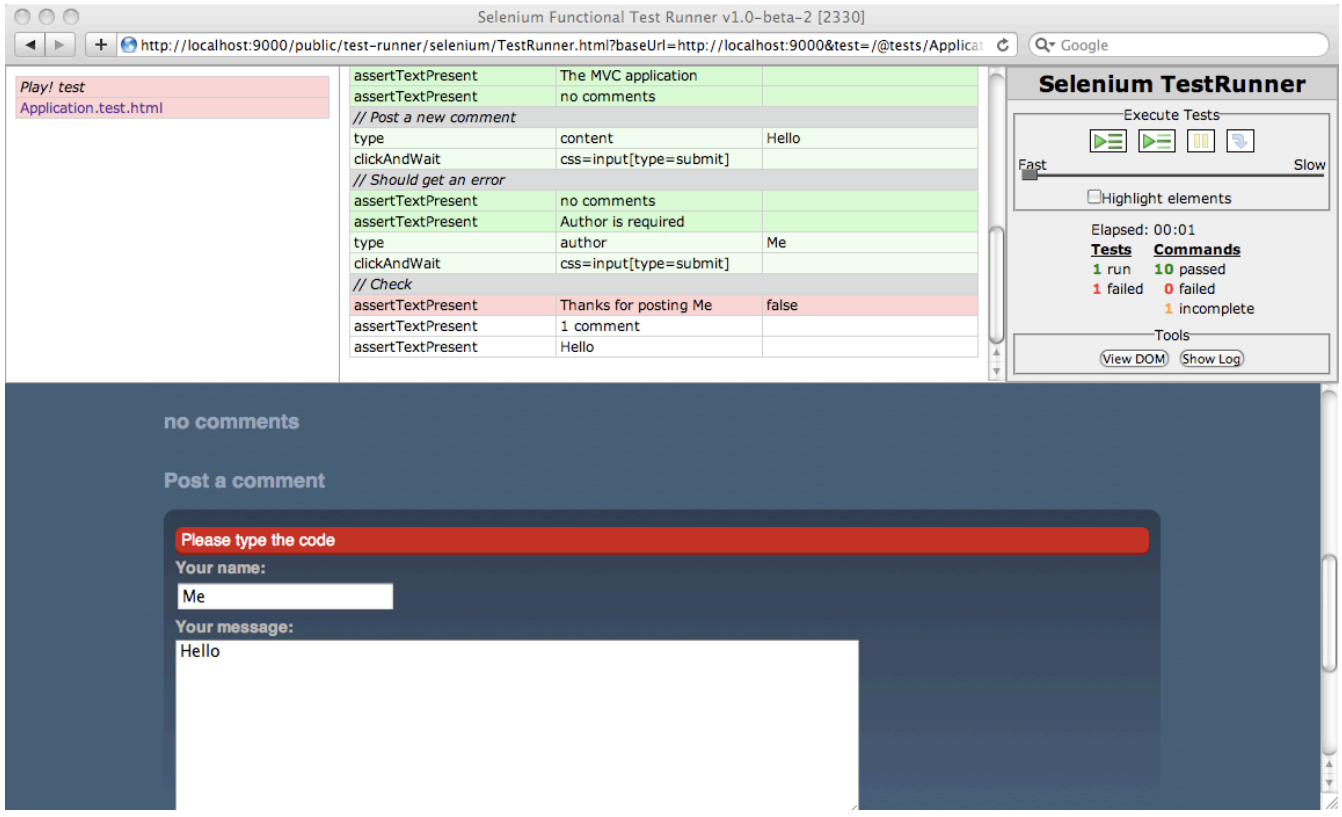
    // Click on 'The MVC application post'
    clickAndWait('link=The MVC application')
    assertTextPresent('The MVC application')
    assertTextPresent('no comments')

    // Post a new comment
    type('content', 'Hello')
    clickAndWait('css=input[type=submit]')

    // Should get an error
    assertTextPresent('no comments')
    assertTextPresent('Author is required')
    type('author', 'Me')
    clickAndWait('css=input[type=submit]')

    // Check
    assertTextPresent('Thanks for posting Me')
    assertTextPresent('1 comment')
    assertTextPresent('Hello')
#{/selenium}
```

And run it. Well it fails; and we have a serious problem here.



We can't really correctly test the captcha mechanism, so we have to cheat. In test mode we will validate any code as a correct captcha. We know that we're in test mode when the framework id is **test**. So let's modify the **postComment** action in the **yabe/app/controllers/Application.java** file to skip this validation in test mode:

```
...
if(!Play.id.equals("test")) {
    validation.equals(code, Cache.get(randomID)).message("Invalid code. Please type it
again");
}
...
```

Now just modify the test case to type any code in the text field, as is:

```
...
type('author', 'Me')
type('code', 'XXXXX')
clickAndWait('css=input[type=submit]')
...
```

And now run the test again, it should work.

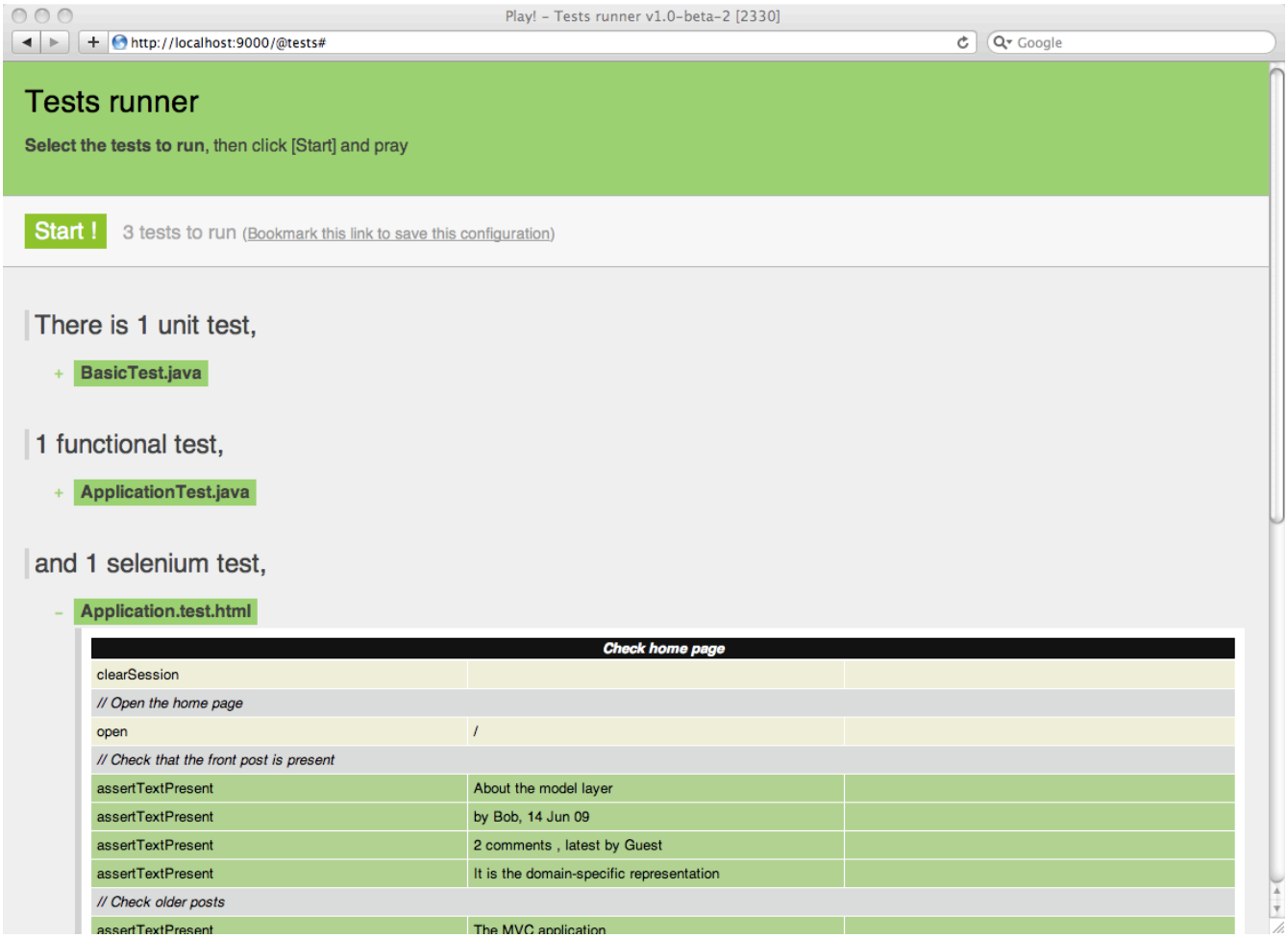
Measuring code coverage

Of course we haven't written all required test cases for the application. But it's enough for this tutorial. Now in a real-world project, how can we know if we have written enough test cases? We need something called **'code coverage'**.

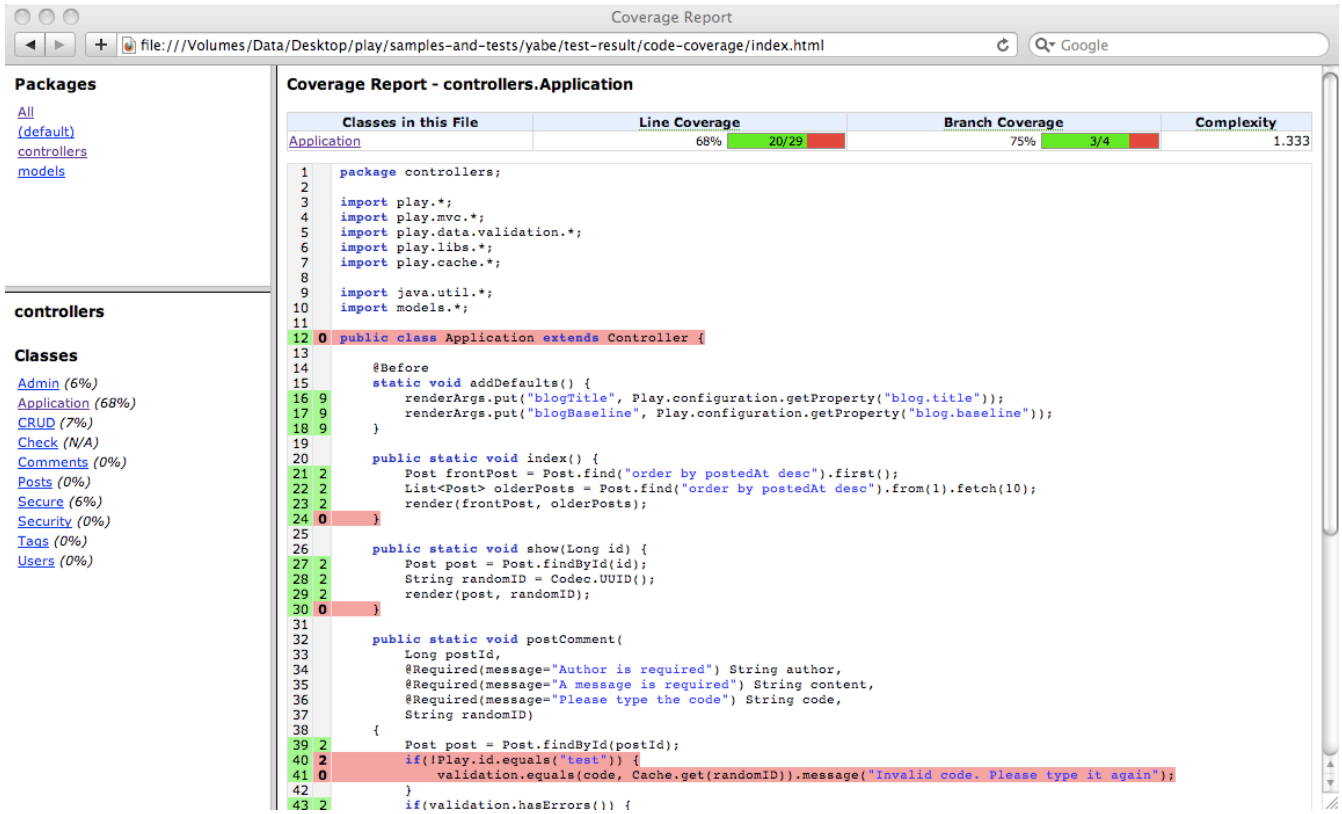
Play comes with a code coverage module based on the Cobertura tool. We need to enable this module only for test mode. So add this line to the **application.conf** file, and restart the application in test mode.

```
# Import the cobertura module in test mode
%test.module.cobertura=${play.path}/modules/cobertura
```

Now reopen the browser at the http://localhost:9000/@tests URL, select all tests and run them. All should be green.



When all tests are passed, stop the application and cobertura will then generate the code coverage report. You can then open the **yabe/test-result/code-coverage/index.html** in your browser and check the report.



As you see we're far from testing all of the application's cases. A good testing suite should approach 100%, even if it is of course nearly impossible to check all the code. Typically because we often need to hack in test mode, like we did for the captcha.

Go to the next part.

This time we've finished the blog engine. Let's see the few common steps we need to put a Play application in production.

Defining a framework ID

You will typically deploy your application on a different computer (a server) than the one you used for development. So you will have a different Play installation.

Play allows to assign each framework installation a different ID, then to manage different configurations in the same **application.conf** file. Let's say that the **server01** will host your production application.

Once the Play framework is installed on this server, let's define the framework ID using the **'play id'** command. Type:

```
$ play id
```

And assign **server01** as id. Now we can define special keys in the yabe configuration file that will be used only when the application is running on the server.

Setting the application in PROD mode

The first configuration key we want specialize for the server deployment is the **application.mode** property. So far we have used the **DEV** mode that allows Play to hot reload and recompile Java files and display detailed messages when an error occurs. In **PROD** mode however, Play will compile all Java sources and templates at startup time and will never check again for changes.

In the **yabe/conf/application.conf** file, define:

```
%server01.application.mode=PROD
```

Now when you run the yabe application on the server, it will automatically start in **PROD** mode.

Configuring a MySQL server

For production use, we will use a MySQL server as database server instead of the in-memory HSQLDB we have used so far. Play comes with the JDBC driver for MySQL so we don't need to install anything more.

Edit the database configuration in the **yabe/conf/application.conf** file:

```
%server01.db=mysql:root:secret@yabe
```

We will now tweak the way Hibernate manages the database schema for us. It's very useful when Hibernate automatically updates the database schema as the Java model objects change.

Change the **jpa.ddl** configuration key:

```
%server01.jpa.ddl=update
```

However it's kind of unpredictable, and running magical things on a production database is never a good thing. So you should always have a backup before a new deployement. If you don't want that hibernate update your database, change the **jpa.ddl** configuration key to validate:

```
%server01.jpa.ddl=validate
```

Setting up a frontal HTTP server

Now we could just change the embedded HTTP server's default port to 80 in order to have a real production server. However it will limit us to installing only one Play application on a given server. As we typically want to install several applications in the same server (but using different IP host names), we need to use a frontal HTTP server as reverse proxy.

You can choose any HTTP server and configure it as reverse proxy, however it is generally a good choice to take something light and fast like LIGHTTPD.

The exact configuration of LIGHTTPD is beyond the scope of this tutorial, but should be something like:

```
server.modules = (
      "mod_access",
      "mod_proxy",
      "mod_accesslog"
)
...
$HTTP["host"] =~ "www.yabe.com" {
    proxy.balance = "round-robin" proxy.server = ( "/" =>
        ( ( "host" => "127.0.0.1", "port" => 9000 ) ) )
}
```

And then allow a local reverse proxy to connect to your Play application by adding this key to the **application.conf** file:

```
%server01.XForwardedSupport=127.0.0.1
```

This is just the beginning

If you have read and followed this tutorial, you're now a seasoned Play developer. You know most of the concepts that drive Play application development.

At this stage, you may want to continue the tutorial with internationalisation and localisation.

There are still more features we haven't explored yet, especially related to Web services, like JSON or XML. Also Play comes with more modules that provide more features. And Play itself still evolves every day.

If you are convinced that Play will help you save time for your next Java web application, you're now ready to start. And don't hesitate to ask us any question on the Play Google Group.

Thank you!

Having built our fully-functional blog engine, we will now consider an optional extra: web application internationalization and language localisation. Although we could have done this from the start, it is more realistic to build the first version of an application in a single language, and add multiple languages later.

Internationalisation and localisation

There are two steps to perform: internationalisation and localisation. Both are mostly about text.

Internationalisation, in programming terms, is a refactoring to remove locale-specific code from the application code. In a web application, this is almost entirely about replacing user-interface text in view templates with references to messages. It also includes formatting non-text data types: dates, currency and other numbers.

Localisation is making a locale-specific version of an application. If the application is internationlised, this means having one or more selectable locale-specific versions. In a web application, this localisation is mostly about translating the user-interface text into the chosen natural language. Language selection is typically a combination of language preferences set in the web browser, and a language selection user-interface in the application itself.

In practice, the two steps go together: you both internationalise and localise one part of the application at a time.

Yet Another Blog Engine

> The starting point for this section is the finished tutorial code, which you can find in the Play distribution's **samples-and-tests/yabe** directory. The goal is to fully internationalise the application, and add French and Dutch language localisations.

To get started, first edit **conf/application.conf** and uncomment (in the default configuration file) or add a line with three supported languages:

```
# Localisations for English, Dutch and French.
application.langs=en,nl,fr
```

If you load a page in the application now, the Play console will show three warnings because you do not have any locale-specific message files yet:

```
16:19:04,728 WARN ~ Messages file missing for locale en
16:19:04,729 WARN ~ Messages file missing for locale nl
16:19:04,729 WARN ~ Messages file missing for locale fr
```

UTF-8 message files

The warnings above mean that you need to replace the existing **conf/messages** file with one message file for each language:

- messages.en
- messages.nl
- messages.fr

At this point we encounter the first improvement over the normal Java way of doing things. These files use the same syntax as Java properties files, but they are not properties files because they must use UTF-8 encoding. Java Properties , on the other hand, specifies ISO-8859-1 'Latin-1' character encoding for streaming to and from text files.

Being able to use UTF-8 message files is a big deal for language localisation, because it means that you can write localised language messages in 'plain text'. For example, this means that for a Greek localisation, instead of:

```
hello.morning =
\u0152\u222b\u0152\u00b1\u0152\u00aa\u0152\u2211\u0152\u00ba\u0152\u2260\u0153\u00c5\u01
= \u0152\u2265\u0152\u00b5\u0152\u03c0\u0152\u00b1
\u0153\u00c9\u0152\u00f8\u0153\u00d6
```

you can use Greek letters instead of those Unicode character escapes:

```
hello.morning =
hello.informal =
```

For the rest of this tutorial, code samples will either define messages in one of these files, or show internationalised mark-up in one of the HTML view templates.

Simple messages

The simple case is a text string that does not change, and that is not interrupted by other markup. For example, the first such text in the **yabe/app/views/main.html** template, in the **tools** list:

```
<ul id="tools">
<li>
<a href="@{Admin.index()}">Log in to write something</a>
</li>
</ul>
```

To internationalise this, we replace the text with a message look-up, using the **&{'key'** } syntax:

```
<ul id="tools">
<li>
<a href="@{Admin.index()}">&{'views.main.tools.login'}</a>
</li>
</ul>
```

To add the localisations, add the corresponding line in each of the three message files. In **conf/messages.en**

```
views.main.tools.login = Log in to write something
```

In **conf/messages.nl**

```
views.main.tools.login = Inloggen om iets te schrijven
```

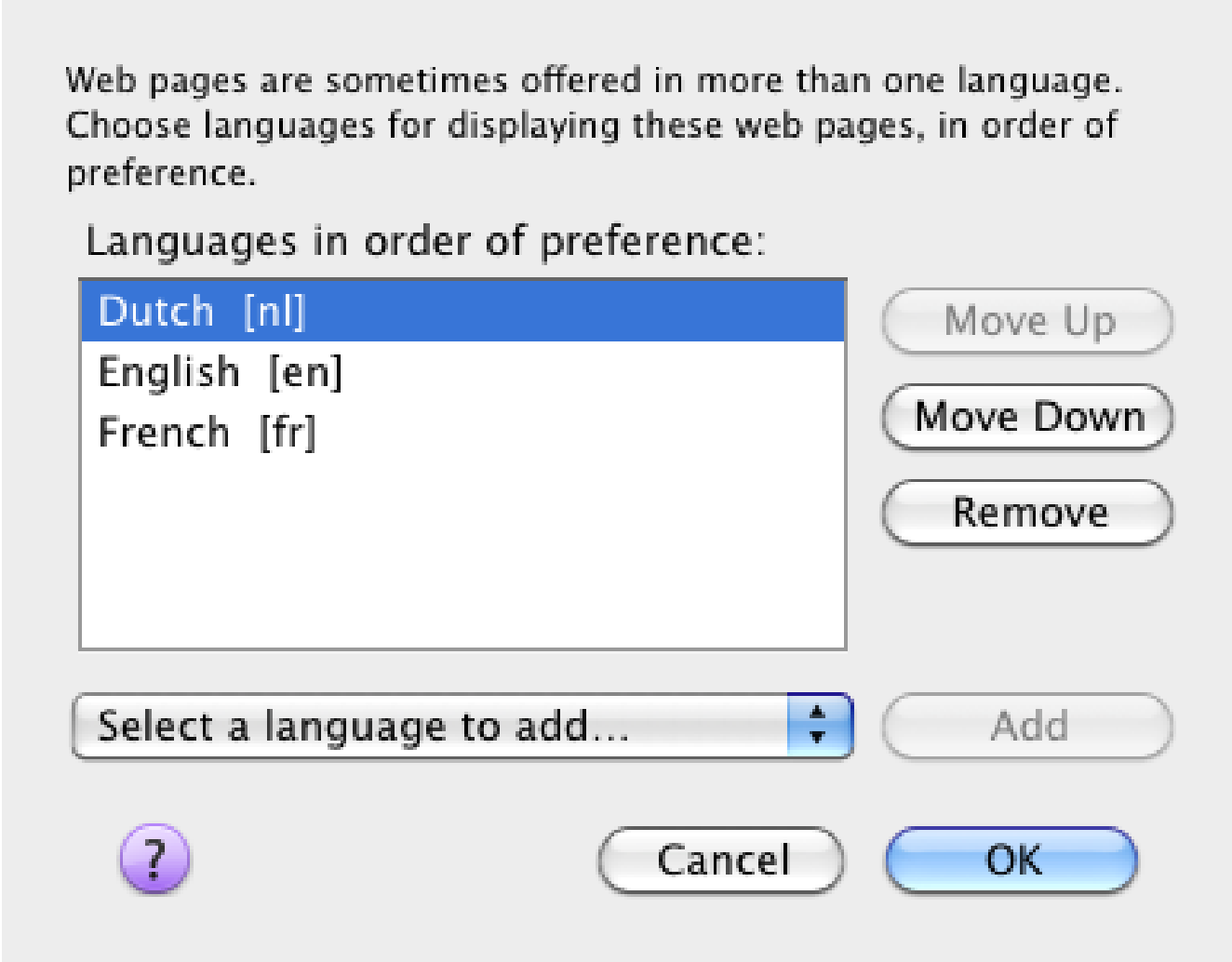
In **conf/messages.fr**

```
views.main.tools.login = Connectez-vous pour écrire quelque chose
```

The message key can be anything you like; in this example I have used a key to indicate the location **views/main.html#tools**

Once you have saved these changes, you can see the different language versions in your web browser by changing the setting that results in a different **Accept-Language** HTTP request header. In Firefox, select Preferences »

Content » Languages » Choose… , add French \[fr] and Dutch \[nl] if they are not already in the list, change which one is at the top of the list, close the dialogue box and reload the page.



Application model localisation

If you use that link to log in to the blog's 'admin' pages, you can access lists of posts, tags, comments and users. These pages are provided by the CRUD module. For each of these pages, the title (light pink) and the column headers are terms from the application's model, i.e. JavaBean class and property names.

The CRUD module internationalises these names using the JavaBean class or property name as the message key, which means you can localise them with messages such as the following.

In **conf/messages.nl**

```
post = artikel
Post = Artikel
posts = artikelen
Posts = Artikelen
comment = reactie
Comment = Reactie
comments = reacties
Comments = Reacties
user = gebruiker
User = Gebruiker
users = gebruikers
Users = Gebruikers
```

In **conf/messages.fr**

```
post = article
Post = Article
posts = articles
Posts = Articles
comment = commentaire
Comment = Commentaire
comments = commentaires
Comments = Commentaires
user = utilisateur
User = Utilisateur
users = utilisateur
Users = Utilisateurs
```

You will notice that this does not change the rounded purple navigation links:



Those are defined **views/admin.html** which you can internationalise to use the same localisations simply by surrounding the existing text with **&{'...'** } as follows:

```
<a href="@{Posts.list()}">&{'Posts'}</a>
…
<a href="@{Tags.list()}">&{'Tags'}</a>
…
<a href="@{Comments.list()}">&{'Comments'}</a>
…
<a href="@{Users.list()}">&{'Users'}</a>
```

Parameterised messages

As well as simple messages, our application includes messages that contain a variable, such as Posts tagged with Play

To localise a message that contains a single parameter, use a Java format string to insert the parameter value in the message:

```
views.Application.listTagged.title = Posts tagged with %s
```

and in the template, add the parameter like this:

```
**&{'views.Application.listTagged.title', tag**}
```

When a message contains multiple parameters, add an index to the format string to allow for different word order in another language:

```
views.Admin.index.welcome = Welcome %1s, <span>you have written %2s posts so
far</span>
```

… with a list in the template:

```
**&{'views.Admin.index.welcome', user, posts.size()**}
```

In this example, we would also like to use the correct plural form for the word 'post', so make that word a parameter too:

```
views.Admin.index.welcome = Welcome %1s, <span>you have written %2s %3s so
far</span>
```

… and use the **pluralize** extension in the template

```
**&{'views.Admin.index.welcome', user, posts.size(),
posts.pluralize(messages.get('post'), messages.get('posts'))**}
```

Note that we have to use **messages.get** to look up the localised singular and plural.

Play module localisation

Play module localisation works the same was as localisation within your application. This application uses the CRUD and Secure modules, which means that we must localise the messages in **play/modules/crud/conf/messages** and **play/modules/secure/conf/messages** that our application uses.

In **conf/messages.nl**

```
# play/modules/crud (administration)
crud.title = Beheer
crud.home = Home
crud.blank = Nieuw
crud.index.title = Kies het te bewerken object
crud.index.objectType = Type object
crud.index.action =
crud.index.add = Voeg toe
crud.add = &{%s} toevoegen
crud.list.title = &{%s}
crud.list.size = %d &{%s}
crud.list.totalSize = %d totaal
crud.pagination.previous = « Vorige
crud.pagination.next = Volgende »
crud.pagination.last = Laatste »»
crud.pagination.first = «« Eerste
crud.show.title = &{%s} bewerken
crud.save = Opslaan
crud.saveAndContinue = Opslaan en verder bewerken
crud.cancel = Annuleren
crud.hasErrors = Corrigeer fouten a.u.b.
crud.blank.title = &{%s} toevoegen
crud.saveAndAddAnother = Opslaan en nogmaals creëren
crud.delete = &{%s} verwijderen
crud.created = &{%s} is aangemaakt
crud.saved = &{%s} is opgeslagen
crud.deleted = &{%s} is verwijderd
crud.delete.error = Kan dit object niet verwijderen
crud.search = Zoeken
crud.none = (Geen)
crud.help.required = Verplicht.
crud.help.minlength = Min. lengte is %d.
crud.help.maxlength = Max. lengte is %d.
crud.help.email = Geldig e-mailadres
crud.help.dateformat = In de vorm YYYY-MM-DD.
crud.help.numeric = Numeriek.
crud.help.min = Moet groter daan %d zijn.
crud.help.future = In de toekomst.
crud.help.past = In het verleden.
crud.help.after = Na %s.
crud.help.before = Voor %s.
crud.help.range = Tussen %d en %d

# play/modules/secure
secure.username = Uw e-mailadres:
secure.password = Uw wachtwoord:
secure.signin = Nu inloggen
```

In **conf/messages.fr**

```
# play/modules/crud (administration)
crud.title = Administration
crud.home = Home
crud.blank = Nouveau
crud.index.title = Choisissez l'objet à modifier
crud.index.objectType = Type objet
crud.index.action = XXX
crud.index.add = Ajouter
crud.add = Ajouter &{%s}
crud.list.title = &{%s}
crud.list.size = %d &{%s}
crud.list.totalSize = %d total
crud.pagination.previous = « Précédent
crud.pagination.next = Suivant »
crud.pagination.last = Dernier »»
crud.pagination.first = «« Premier
crud.show.title = Modifier &{%s}
crud.save = Enregistrer
crud.saveAndContinue = Enregistrer et continuez à modifier
crud.cancel = Annuler
crud.hasErrors = Corrigez les erreurs s.v.p.
crud.blank.title = Ajouter &{%s}
crud.saveAndAddAnother = Enregistrer et ajouter un autre
crud.delete = Supprimer &{%s}
crud.created = &{%s} a été crée
crud.saved = &{%s} est enregistré
crud.deleted = &{%s} est supprimé
crud.delete.error = Ne peut pas supprimer l'objet
crud.search = Chercher
crud.none = (aucun)
crud.help.required = Obligatoire.
crud.help.minlength = Longeur minimum est %d.
crud.help.maxlength = Longeur maximum est %d.
crud.help.email = Adresse e-mail valide
crud.help.dateformat = En format YYYY-MM-DD.
crud.help.numeric = Numerique.
crud.help.min = Doit être plus grand que %d.
crud.help.future = Dans le futur.
crud.help.past = Dans le passé.
crud.help.after = Après %s.
crud.help.before = Avant %s.
crud.help.range = Entre %d et %d

# play/modules/secure
secure.username = Votre adresse e-mail:
secure.password = Votre mot de passe:
secure.signin = Connectez-vous maintenant
```

Of course, once you have done this it is also a good idea to contribute the localisations back to the module.

Special cases

When you are localising a web application, there are a few special cases that be awkward to implement if you are using a component-based web application framework, such as JavaServer Faces:

1. Parameterised message used in an attribute value
2. Formatted message parameter
3. Link within message

All three cases turn out to be straightforward in Play.

The first case happens when you want to use a phrase with a parameter in an attribute value in the template, such as:

```
<a href="@{Application.show(_post.id)}" title="By Bob">
```

This is a problem in JSF, because you would normally use an XML tag to perform the parameter replacement, which you cannot do in an attribute value. The Play syntax simply avoids this problem, and you can just do:

```
<a href="@{Application.show(_post.id)}" title="&{'views.tags.display.author',
_post.author.fullname}">
```

The second case is when you want to format a value, such as a date, for use as a message parameter in a phrase like **By Bob on 2009-06-14**. Again, the problem in JSF is caused by having to an XML tag to format the value, while needing to be able to use the result in an XML attribute value. In Play the formatting extensions do not get in the way of the message parameter syntax, so you can do:

```
<span>&{'views.tags.display.author', _post.author.fullname,
comment.postedAt.format('yyyy-MM-dd')}"}</span>
```

You can, of course, localise the format pattern as well:

```
<span>&{'views.tags.display.author', _post.author.fullname,
comment.postedAt.format(messages.get('views.dateFormat'))}"}</span>
```

The third case typically occurs when you want part of a localised message to be a hyperlink, as in the message **Log in to write something**. **This is a problem in JSF because the hyperlink is a JSF component that is rendered in a way that means the link's mark-up cannot be in the message file. Play on the other hand, lets you use plain HTML in your templates, so you can just put the mark-up in your message with a parameter for the URL:**

```
logIn = <a href="%s">Log in</a> to write something
```

```
&{'logIn', '/admin'}
```

Our application was using the

```
<a href="@{Admin.index()}">
```

syntax in the hyperlink to get the framework to generate the URL based on the routes file. To do this in the message parameter, do:

```
&{'logIn', actionBridge.Admin.index()}
```

The localised 'Yet Another Blog Engine' example

The end result of applying the above steps is a localised version of the 'Yet Another Blog Engine' example that works in English, Dutch and French.



The 'Yet Another Blog Engine' admin interface in Dutch (above) and French (below).



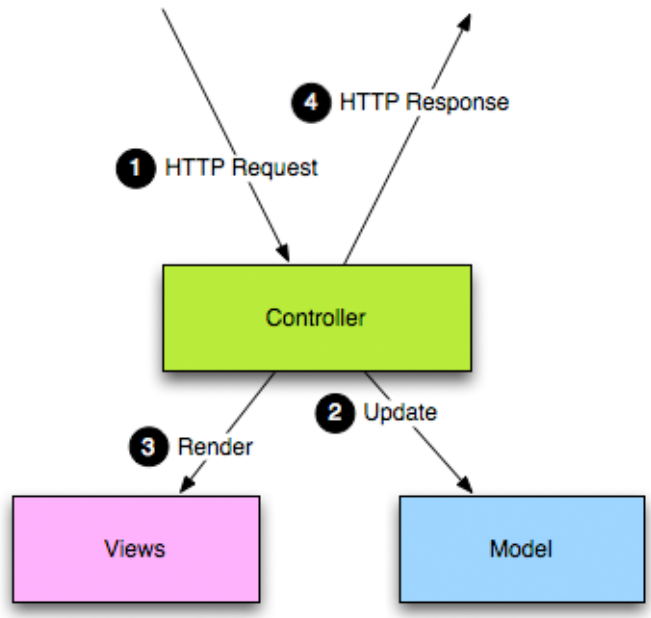
By Peter Hilton, originally published on the Lunatech Research blog.

The MVC application model

A Play application follows the MVC architectural pattern applied to the web architecture.

This pattern splits the application into separate layers: the Presentation layer and the Model layer. The Presentation layer is further split into a View and a Controller layer.

- The Model is the domain-specific representation of the information on which the application operates. Domain logic adds 'meaning' to raw data (e.g., calculating if today is the user's birthday, or the totals, taxes, and shipping charges for a shopping cart). Most applications use a persistent storage mechanism such as a database to store data. MVC does not specifically mention the data access layer because it is understood to be underneath, or encapsulated by, the Model.
- The View renders the model into a form suitable for interactions, typically a user interface. Multiple views can exist for a single model, for different purposes. In a Web application the view is usually rendered in a 'web format' like HTML, XML or JSON. However there are some cases where the view can be expressed in a binary form, e.g. dynamically rendered chart diagrams.
- The Controller responds to events (typically user actions) and processes them, and may also invoke changes on the model. In a Web application, events are typically HTTP requests: a Controller listens for HTTP requests, extracts relevant data from the 'event', such as query string parameters, request headers… and applies changes to the underlying model objects.



In a Play application these three layers are defined in the **app** directory, each one in a separate Java package.

app/controllers

A Controller is a Java class where each public, static, method is an **action**. An action is a Java entry point invoked when an HTTP Request is received. The Java code from the Controller class isn't really object oriented: it's mainly procedural code. The action method extracts relevant data from the HTTP Request, reads or updates the model objects, and sends back a result which is wrapped into an HTTP Response.

app/models

The domain model object layer is a set of Java classes using all the object-oriented features available from the Java language. It contains data structures and operations on which the application operates. Whenever model objects need to be saved to persistent storage, they may contain some glue artifacts like JPA annotations or SQL statements.

app/views

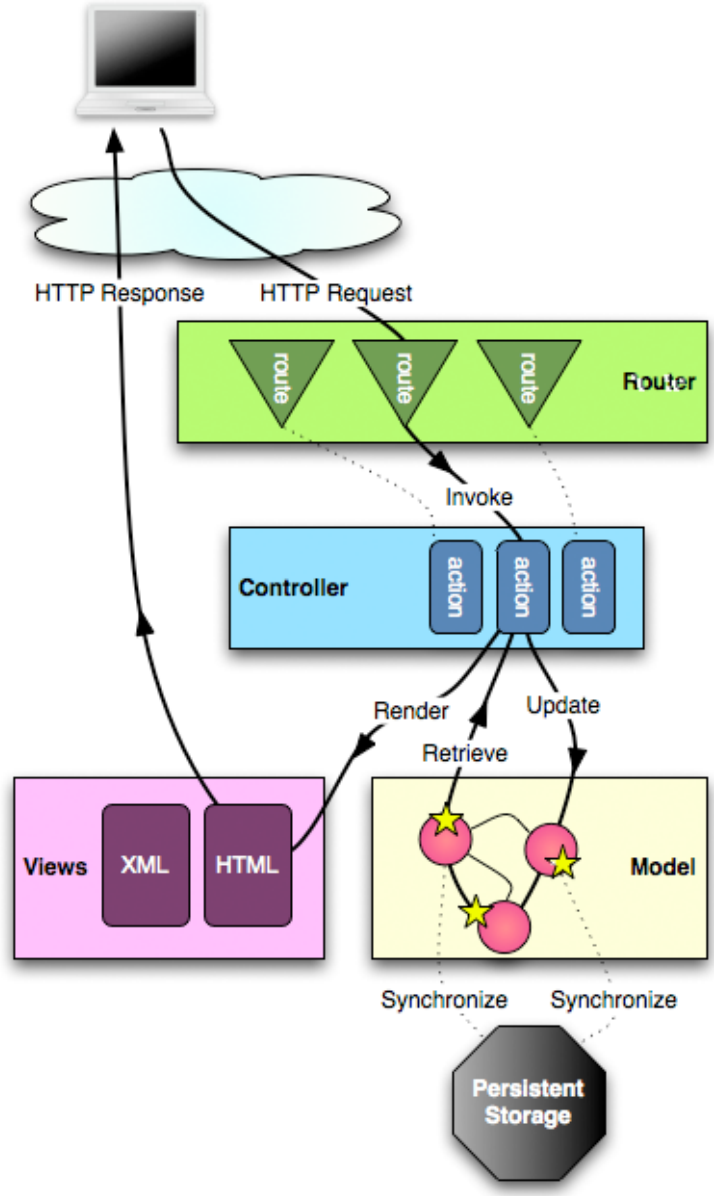
Most of the application views are generated using an efficient templating system provided by Play. The Controller gets some interesting data from the model layer, and then applies a template to decorate these objects. This package contains HTML, XML, JSON or other template files with special directives used to dynamically generate the model representation.

The request life-cycle

The Play framework is fully stateless and only request/response-oriented. All HTTP Requests follow the same path:

1. An HTTP Request is received by the framework.
2. The Router component tries to find the most specific route able to accept this request. The corresponding action method is then invoked.
3. The application code is executed.
4. If a complex view needs to be generated, a template file is rendered.
5. The result of the action method (HTTP Response code, Content) is then written as an HTTP Response.

The following diagram summarizes the HTTP Request path:



The standard application layout

The layout of a Play application is standardized to keep things as simple as possible.

The app directory

This directory contains all executable artifacts: Java source code and view templates.

Where are my .class files?

Don't look for compiled Java classes. The framework compiles the Java source code at runtime and only keeps compiled classes in a bytecode cache under the **tmp** directory. The main executable artifacts in a Play application are the **.java** source files, not the compiled classes.

There are three standard packages in the app directory, one for each layer of the MVC architectural pattern. You can of course add your own packages like for example a **utils** package.

In addition, the views package is further organized into sub-packages:

- tags, hosts application tags, e.g. reusable pieces of templates.
- One views folder for each Controller, by convention templates related to each Controller are stored in their own sub-package.

The public directory

Resources stored in the **public** directory are static assets and are served directly by the Web server.

This directory is split into three standard sub-directories: for images, CSS stylesheets and JavaScript files. You should try to organize your static assets like this to keep all Play applications consistent.

Tip

By default the /public directory is mapped to the /public URL path, but you can easily change that, or even use several directories for your static assets.

The conf directory

The **conf** directory contains all configuration files for the application.

There are two required configuration files:

- application.conf, the main configuration file for the application. It contains standard configuration options.
- routes, the routes definition file.

Tip

If you need to add some configuration options specific to your application, it's a good idea to add more options to the application.conf file. If any library needs a specific configuration file, try to file it under the conf directory: **this directory is included in the Java ClassPath**.

The lib directory

This directory contains all standard Java libraries needed by your application. They are automatically added to the Java classpath.

Development life cycle

There are no compilation, packaging or deployment phases while working with Play. However Play implements two distinct environments: DEV mode during the development phase and PROD mode when the application is deployed.

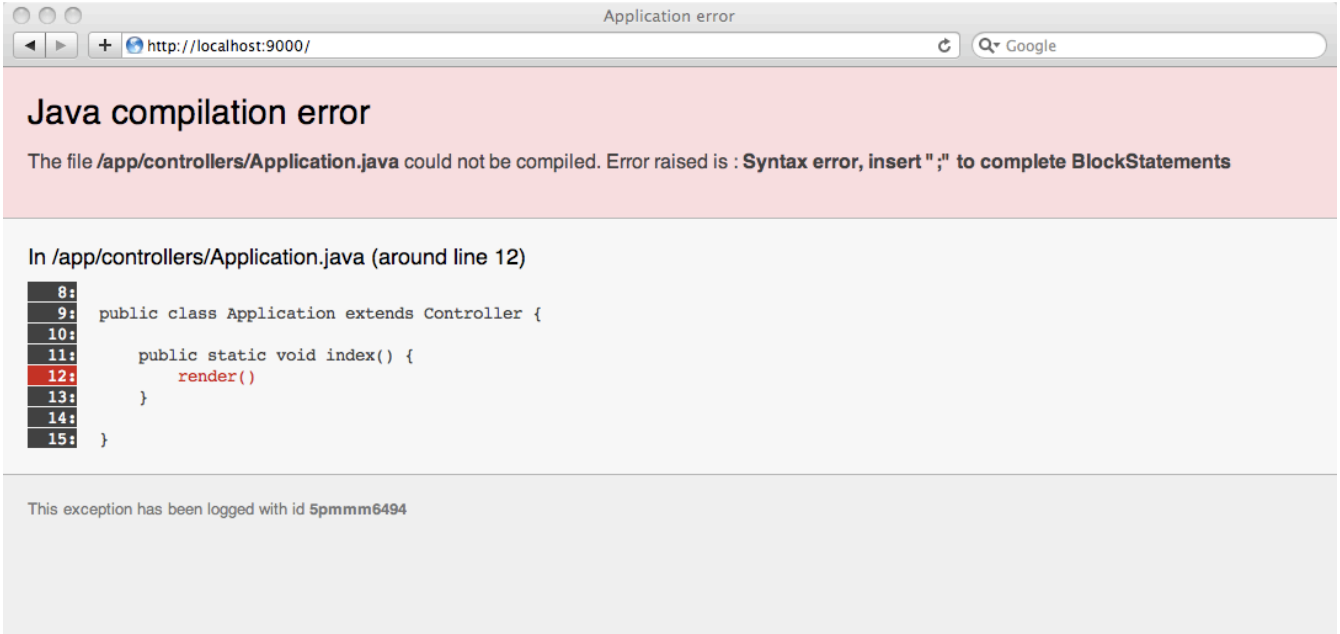
About DEV/PROD modes

You can run an application either in a DEV or PROD mode. You toggle this mode using the **application.mode** configuration property. When run in DEV mode, Play will check for file changes and will handle hot reloading if necessary.

The PROD mode is fully optimized for production: Java sources and templates are compiled once and cached for multiple uses.

Java source code is compiled and loaded at runtime. If a Java source file is modified while the application is running, the source code is recompiled and hot-swapped into the JVM.

If a compilation error occurs, the exact problem is displayed in the browser (in DEV mode only).

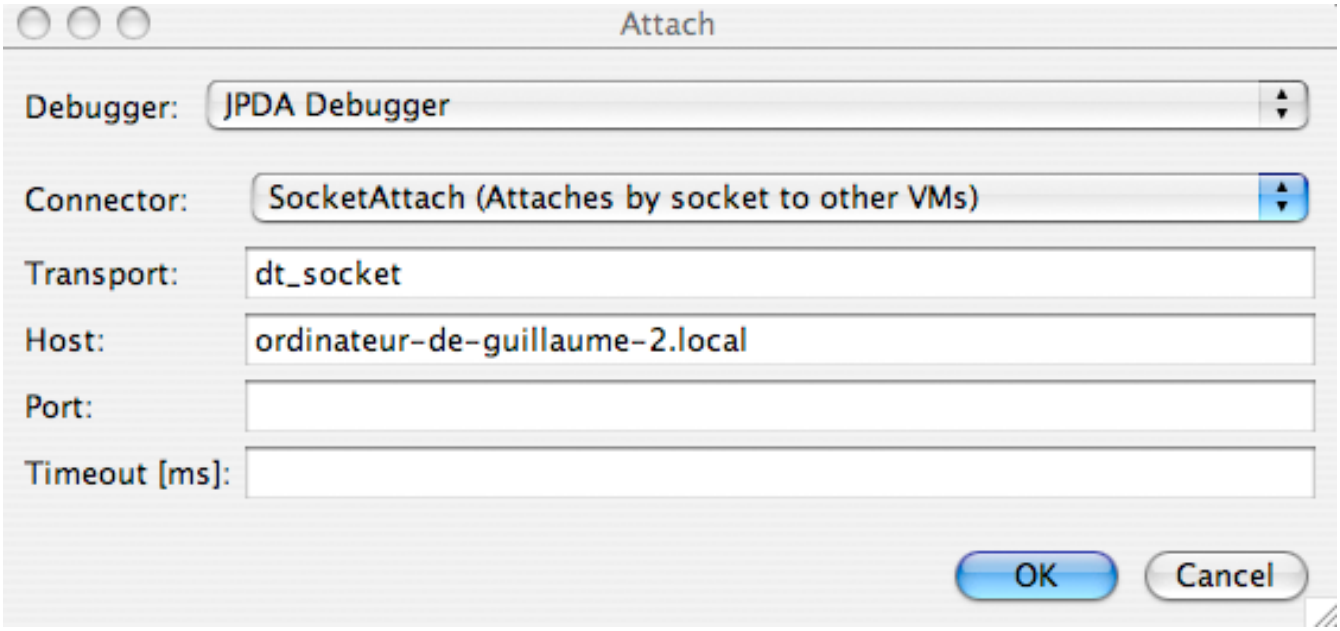


Template files are hot-compiled and hot-reloaded too.

Connect a Java debugger

When you run the application in DEV mode, you can connect a Java debugger to the port 8000.

For example, using the NetBeans debugger:



Continuing the discussion

Now that you've seen what a Play application is, let's see how the Router works. The Router is in charge of translating incoming HTTP Requests into actions.

The router is the component in charge of translating incoming HTTP Requests into action calls (a static, public method of a Controller).

An HTTP request is seen as an event by the MVC framework. The event contains two major pieces of information:

- The Request path (such as /clients/1542, /photos/list), including the query string.
- The HTTP method (GET, POST, PUT, DELETE)

About REST

Representational state transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web.

REST states a few key design principles:

- Application functionality is divided into resources
- Every resource is uniquely addressable using a URI
- All resources share a uniform interface for the transfer of state between client and resource.

If you're using HTTP, these interfaces are defined by the set of available HTTP methods. The protocol used to access the resource state is:

- Client-server
- Stateless
- Cacheable
- Layered

If an application follows the main REST design principles, the application is RESTful. The Play framework makes it easy to build RESTful applications:

- The Play router interprets both URI and HTTP methods to route a request to a Java call. Regular expressions-based URI patterns give you even more flexibility.
- The protocol is stateless. This means you can't save any state on the server between two successive requests.
- Play considers HTTP as a key feature, thus the framework gives you full access to HTTP information.

The routes file syntax

The **conf/routes** file is the configuration file used by the Router. This file lists all the routes needed by the application. Each route consists of an HTTP method + URI pattern associated with a Java call.

Let's see what a route definition looks like:

```
GET     /clients/{id}             Clients.show
```

Each route starts with the HTTP method, followed by the URI pattern. The last element of a route is the Java call definition.

You can add a comment to the route file, with the **"#"** character.

```
# Display a client
GET     /clients/{id}             Clients.show
```

The HTTP method

The HTTP method can be any of the valid methods supported by HTTP:

- GET
- POST
- PUT
- DELETE
- HEAD

If you specify * as method, this route will match the HTTP Request for any method.

```
*   /clients/{id}             Clients.show
```

This route will accept both of:

```
GET /clients/1541
PUT /clients/1212
```

The URI Pattern

The URI pattern defines the route's request path. Some parts of the request path can be dynamic. Any dynamic part must be specified within braces {…}.

```
/clients/all
```

exactly matches:

```
/clients/all
```

but…

```
/clients/{id}
```

matches both:

```
/clients/12121
/clients/toto
```

A URI pattern may have more than one dynamic part:

```
/clients/{id}/accounts/{accountId}
```

The default matching strategy for a dynamic part is defined by the regular expression **/[^/]+/**. You can define your own regular expression for a dynamic part.

This regex will only accept numerical values as id:

```
/clients/{<[0-9]+>id}
```

This one will ensure id is a word containing between 4 and 10 lower case characters only:

```
/clients/{<[a-z]{4,10}>id}
```

Any valid regular expression can be used here.

Note

Dynamic parts are named. The Controller can later retrieve the dynamic parts from the HTTP params map.

By default Play considers the trailing URL slash as important. For example, this route:

```
GET     /clients         Clients.index
```

will match the **/clients** URL but not **/clients/**. You can tell Play that you want to match both URLs by adding a question mark after the trailing slash. For example:

```
GET     /clients/?       Clients.index
```

The URI pattern cannot have any optional part except for that trailing slash.

Java call definition

The last part of a route definition is the Java call. This part is defined by the fully-qualified name of an action method. The action method must be a static, public method of a Controller class. A Controller class must be defined in the **controllers** package and must be a subclass of **play.mvc.Controller**.

You can add a Java package before the Controller class name if it isn't defined directly under the controllers package. The **controllers** package itself is implicit, so you don't need to specify it.

```
GET    /admin             admin.Dashboard.index
```

Assign static args

In some cases, you want to reuse an existing action but define a more specific route based on the values of some of the arguments.

Let's see how in this example:

```
public static void page(String id) {
    Page page = Page.findById(id);
    render(page);
}
```

With the corresponding route:

```
GET    /pages/{id}        Application.page
```

Now, I want to define a URL alias for the page with ID 'home'. I can define another route with a static argument:

```
GET    /home              Application.page(id:'home')
GET    /pages/{id}        Application.page
```

The first route is equivalent to the second one when the page ID is 'home'. However, since it has higher priority, this route will be used as the default for the call to Application.page with ID 'home'.

Routes priority

Many routes can match the same request. If there is any conflict, the first route (following the declaration order) is used.

For example:

```
GET    /clients/all       Clients.listAll
GET    /clients/{id}      Clients.show
```

With these definitions, the URI:

```
/client/all
```

will be intercepted by the first route and will call Clients.listAll (even if the second route matched the request too).

Serving static resources

Use the special action **staticDir**, to point to each folder you wish to publish as a static resources container.

For example:

```
GET    /public/           staticDir:public
```

When supplied with a request for a /public/* path, Play will serve your files from the application /public folder.

Priorities are applied as for a standard route.

Reverse routing: generate some URL

The Router can be used to generate a URL from within a Java call. So you're able to centralize in one only configuration file all your URI patterns, and then be more confident when refactoring your application.

For example, with this route definition:

```
GET    /clients/{id}      Clients.show
```

From your code, you can generate the URL able to invoke Clients.show:

```
map.put("id", 1541);
String url = Router.reverse("Clients.show", map).url;  // GET /clients/1541
```

> The URL generation is integrated into many of the framework's components. You never should use the Router.reverse operation directly.

If you add parameters that are not included in the URI pattern, these parameters will be added to the query string:

```
map.put("id", 1541);
map.put("display", "full");
String url = Router.reverse("Clients.show", map).url; // GET
/clients/1541?display=full
```

The priority order is again used to find the most specific Route able to generate the URL.

Setting content types

Play selects a media type for the HTTP response according to the value of **request.format**. This value determines which view template file to use, by file extension, and also sets the response **Content-type** to the media type that Play's **mime-types.properties** file maps the format to.

The default format for a Play request is **html**. The default template for the **index()** controller method (and **html** format) is therefore the file **index.html**. If you specify a different format, in one of several ways, you can select an alternate template.

You can set the format programmatically before calling the **render** method. For example, to serve a Cascading Style Sheet with media type **text/css**, you can do:

```
request.format = "css";
```

However, a cleaner approach is to use the URL to specify the format in the **routes** file. You can add formats as specific routes, by specifying the format for the controller method. For example, the following route will handle a request for **/index.xml**, setting the format to **xml** and rendering the **index.xml** template.

```
GET    /index.xml         Application.index(format:'xml')
```

Similarly:

```
GET    /stylesheets/dynamic_css   css.SiteCSS(format:'css')
```

Play can also extract the format directly from the URL, with a route such as the following.

```
GET    /index.{format}    Application.index
```

With this route, a request for **/index.xml** will set the format to **xml** and render the XML template, while **/index.txt** will render the plain text template.

Play can also set the format automatically using HTTP content negotiation.

HTTP content negotiation

One thing that Play has in common with other RESTful architectures is the direct use of HTTP functionality, instead of trying to hide HTTP or put an abstraction layer on top of it. Content negotiation is an HTTP feature that allows an HTTP server to serve different media types for the same URL, according to which media types are requested by the HTTP client. The client specifies acceptable content types using media types in the **Accept** header, such as requiring an XML response with:

```
Accept: application/xml
```

A client may specify more than one media type, and also specify that any media type is acceptable with a catch-all wild-card media type (***/***):

```
Accept: application/xml, image/png, */*
```

Conventional web browsers always include the wild-card value in the **Accept** header: they will accept any media type, and Play will serve HTML - the default 'format'. Content negotiation is more likely to be used by custom clients, such as an Ajax request that requires a JSON response, or an e-book reader that requires a PDF or EPUB version of a document.

Setting the content type from HTTP headers

Play selects its default request format, **html**, if the **Accept** header contains **text/html** or **application/xhtml**, or as a result of the wildcard ***/*** value. The default format is not selected if the wildcard value is not present.

Play has built-in support for a few formats: **html**, **txt**, **json** and **xml**. For example, define a controller method that renders some data:

```
public static void index() {
   final String name = "Peter Hilton";
   final String organisation = "Lunatech Research";
   final String url = "http://www.lunatech-research.com/";
   render(name, organisation, url);
}
```

If you request a URL that is mapped to this method (**http://localhost:9000/** in a new Play application) in a web browser, then play will render the **index.html** template, because web browsers send an **Accept** header that includes the value **text/html**.

Play responds to a request with the header **Accept: text/xml** by setting the request format to **xml** and rendering an **index.xml** template, such as:

```
<?xml version="1.0"?>
<contact>
<name>${name}</name>
<organisation>${organisation}</organisation>
<url>${url}</url>
</contact>
```

The built in **Accept** header format mappings work as follows, for an **index()** controller method: the accept header contains a media type that Play maps to a format, which is in turn mapped to a template file.

Accept header	Format	Template file name	Mapping
null	null	index.html	Default template extension for null format
image/png	null	index.html	Media type not mapped to a format
/, image/png	html	index.html	Default media type mapped to html format
text/html	html	index.html	Built-in format
application/xhtml	html	index.html	Built-in format
text/xml	xml	index.xml	Built-in format
application/xml	xml	index.xml	Built-in format
text/plain	txt	index.txt	Built-in format

text/javascript	json	index.json	Built-in format
application/json, */*	json	index.json	Built-in format, default media type ignored

Custom formats

You can add content negotiation for your own custom types by inspecting the request headers and setting the format accordingly, so that you only set that format when the HTTP request selects the corresponding media type. For example, to serve a vCard with media type **text/x-vcard**, in your controller, check for your custom format before all requests:

```
@Before
static void setFormat() {
        if (request.headers.get("accept").value().equals("text/x-vcard")) {
                request.format = "vcf";
        }
}
```

Now, a request with an **Accept: text/x-vcard** header will render an **index.vcf** template, such as:

```
BEGIN:VCARD
VERSION:3.0
N:${name}
FN:${name}
ORG:${organisation}
URL:${url}
END:VCARD
```

Continuing the discussion

When the Router has determined which Java call to invoke for the received HTTP Request, the Play framework then invokes that Java call. Let's see how the Controller works.

Business logic is managed in the domain model layer. As a client (typically a web browser) cannot directly invoke this code, the functionality of a domain object is exposed as resources represented by URIs.

A client uses the uniform API provided by the HTTP protocol to manipulate these resources, and by implication the underlying business logic. However, this mapping of resources to domain objects is not a bijection: the granularity can be expressed at different levels, some resources may be virtual, for some resources aliases may be defined…

This is precisely the role played by the Controller layer: providing a **glue** between the domain model objects and transport layer events. As the Model layer, controllers are written in pure Java, making it easy to access or modify Model objects. Like the HTTP interface, Controllers are procedural and Request/Response oriented.

The Controller layer reduces the **impedance mismatch** between HTTP and the Domain Model.

Note

There are different architectural models with different strategies. Some protocols give you direct access to the domain model objects. This is typically what EJB or Corba protocols do. In these cases, the architectural style used is RPC (Remote Procedure Call). These communication styles are hardly compatible with web architecture.

Some technologies like SOAP try to give access to the model object domain through the Web. However, SOAP is just another RPC-style protocol, in this case using HTTP as a transport protocol. It is not an application protocol.

The web's principles are not fundamentally object-oriented. So a layer is needed to adapt HTTP to your favorite language.

A controller overview

A Controller is a Java class, hosted by the **controllers** package, and subclassing **play.mvc.Controller**.

This is a Controller:

```
package controllers;

import models.Client;
import play.mvc.Controller;

public class Clients extends Controller {

    public static void show(Long id) {
        Client client = Client.findById(id);
        render(client);
    }

    public static void delete(Long id) {
        Client client = Client.findById(id);
        client.delete();
    }

}
```

Each public, static method in a Controller is called an action. The signature for an action method is always:

```
public static void action_name(params...);
```

You can define parameters in the action method signature. These parameters will be automatically resolved by the framework from the corresponding HTTP parameters.

Usually, an action method doesn't include a return statement. The method exit is done by the invocation of a **result** method. In this example, **render(…)** is a result method that executes and displays a template.

Retrieving HTTP parameters

An HTTP request contains data. This data can be extracted from:

- The URI path: in /clients/1541, 1541 is the dynamic part of the URI Pattern.
- The Query String: /clients?id=1541.
- The request body: if the request was sent from an HTML form, the request body contains the form data encoded as x-www-urlform-encoded.

In all cases, Play extracts this data and builds a Map<String, String[]> which contains all the HTTP parameters. The key is the parameter name. The parameter name is derived from:

- The name of the dynamic part of the URI (as specified in the route)
- The name portion of a name-value pair taken from the Query String
- The contents of a x-www-urlform-encoded body.

Using the params map

The **params** object is available to any Controller class (it is defined in the **play.mvc.Controller** super class). This object contains all the HTTP parameters found for the current request.

For example:

```
public static void show() {
    String id = params.get("id");
    String[] names = params.getAll("names");
}
```

You can also ask Play to do the type conversion for you:

```
public static void show() {
    Long id = params.get("id", Long.class);
}
```

But wait, there are better ways to do this :)

From the action method signature

You can retrieve HTTP parameters directly from the action method signature. The Java parameter's name must be the same as the HTTP parameter's.

For example, in this request:

```
/clients?id=1451
```

An action method can retrieve the **id** parameter value by declaring an **id** parameter in its signature:

```
public static void show(String id) {
    System.out.println(id);
}
```

You can use other Java types than String. In this case the framework will try to cast the parameter value to the correct Java type:

```
public static void show(Long id) {
    System.out.println(id);
}
```

If the parameter is multivalued, you can declare an Array argument:

```
public static void show(Long[] id) {
    for(String anId : id) {
        System.out.println(id);
    }
}
```

or even a collection type:

```
public static void show(List<Long> id) {
    for(String anId : id) {
        System.out.println(id);
    }
}
```

Exceptions

If the HTTP parameter corresponding to the action method argument is not found, the corresponding method argument is set to its default value (typically null for objects and 0 for primitive numeric types). If a value is found but can't be properly cast to the required Java type, an error is added to the validation error collection and the default value is used.

Advanced HTTP to Java binding

Simple types

All the native and common Java types are automatically bound:

int, **long**, **boolean**, **char**, **byte**, **float**, **double**, **Integer**, **Long**, **Boolean**, **Char**, **String**, **Byte**, **Float**, **Double**.

Note that if the parameter is missing in the HTTP Request, or if automatic conversion fails, Object types will be set to null and native types will be set to their default values.

Date

A date object can be automatically bound if the date's string representation matches one of the following patterns:

- yyyy-MM-dd'T'hh:mm:ss'Z' // ISO8601 + timezone
- yyyy-MM-dd'T'hh:mm:ss'' // ISO8601
- yyyy-MM-dd
- yyyyMMdd'T'hhmmss
- yyyyMMddhhmmss
- dd'/'MM'/'yyyy
- dd-MM-yyyy
- ddMMyyyy
- MMddyy
- MM-dd-yy
- MM'/'dd'/'yy

Using the @Bind annotation, you can specify the date format.

For example:

```
archives?from=21/12/1980
```

```
public static void articlesSince(@Bind("dd/MM/yyyy") Date from) {
    List<Article> articles = Article.findBy("date >= ?", from);
    render(articles);
}
```

If no **@Bind** annotation is specified, then Play! uses the default date format according to your locale.
To set the default date format to use, edit your application.conf and set the following property:

```
date.format=yyy-MM-dd
    date.format.fr=dd/MM/yyyy
```

Please note that the language fr in the application.conf must be enabled as well:
bc. application.langs=fr

This property also affects how the dates are rendered in the templates using the ${date.format()}.

Calendar

The calendar binding works exactly as with the date, except that Play! is choosing the Calendar object according to your locale. The @Bind annotation can also be used.

Files

File upload is easy with Play. Use a **multipart/form-data** encoded request to post files to the server, and then use the **java.io.File** type to retrieve the file object:

```
public static void create(String comment, File attachment) {
    String s3Key = S3.post(attachment);
    Document doc = new Document(comment, s3Key);
    doc.save();
    show(doc.id);
}
```

The created file has the same name as the original file. It's stored in a temporary directory and deleted at the end of the request. So you have to copy it in a safe directory or it will be lost.

Arrays or collections of supported types

All supported types can be retrieved as an Array or a collection of objects:

```
public static void show(Long[] id) {
    ...
}
```

or:

```
public static void show(List<Long> id) {
    ...
}
```

or:

```
public static void show(Set<Long> id) {
    ...
}
```

POJO object binding

Play also automatically binds any of your model classes using the same simple naming convention rules.

```
public static void create(Client client ) {
    client.save();
    show(client);
}
```

A query string to create a client using this action would look like:

```
?client.name=Zenexity&client.email=contact@zenexity.fr
```

Play creates a Client instance and resolves HTTP parameter names to properties on the Client object. Unresolved parameters are safely ignored. Type mismatches are also safely ignored.

Parameter binding is done recursively, which means you can address complete object graphs:

```
?client.name=Zenexity
&client.address.street=64+rue+taitbout
&client.address.zip=75009
&client.address.country=France
```

In order to update a list of model objects, use array notation and reference the object's ID. For example imagine the Client model has a list of Customer models declared as **List Customer customers**. To update the list of Customers you would provide a query string like the following:

```
?client.customers[0].id=123
&client.customers[1].id=456
&client.customers[2].id=789
```

JPA object binding

You can automatically bind a JPA object using the HTTP to Java binding.

You can provide the **user.id** field yourself in the HTTP parameters. When Play finds the **id** field, it loads the matching instance from the database before editing it. The other parameters provided by the HTTP request are then applied. So you can save it directly.

```
public static void save(User user) {
    user.save(); // ok with 1.0.1
}
```

Result types

An action method has to generate an HTTP response. The easiest way to do this is to emit a Result object. When a Result object is emitted, the normal execution flow is interrupted and the method returns.

For example:

```
public static void show(Long id) {
    Client client = Client.findById(id);
    render(client);
    System.out.println("This message will never be displayed !");
}
```

The **render(…)** method emits a Result object and stops further method execution.

Return some textual content

The **renderText(…)** method emits a simple Result event which writes some text directly to the underlying HTTP Response.

Example:

```
public static void countUnreadMessages() {
    Integer unreadMessages = MessagesBox.countUnreadMessages();
    renderText(unreadMessages);
}
```

You can format the text message using the Java standard formatting syntax:

```
public static void countUnreadMessages() {
    Integer unreadMessages = MessagesBox.countUnreadMessages();
    renderText("There are %s unread messages", unreadMessages);
}
```

Execute a template

If the generated content is complex, you should use a template to generate the response content.

```
public class Clients extends Controller {

    public static void index() {
        render();
    }

}
```

A template name is automatically deduced from the Play conventions. The default template path is resolved using the Controller and action names.

In this example the invoked template is:

```
app/views/Clients/index.html
```

Add data to the template scope

Often the template needs data. You can add these data to the template scope using the **renderArgs** object:

```
public class Clients extends Controller {

    public static void show(Long id) {
        Client client = Client.findById(id);
        renderArgs.put("client", client);
        render();
    }

}
```

During template execution, the **client** variable will be defined.

For example:

```
<h1>Client ${client.name}</h1>
```

A simpler way to add data to the template scope

You can pass data directly to the template using **render(...)** method arguments:

```
public static void show(Long id) {
    Client client = Client.findById(id);
    render(client);
}
```

In this case, the variables accessible by the template have the same name as the local Java variables.

You can pass more than one variable:

```
public static void show(Long id) {
    Client client = Client.findById(id);
    render(id, client);
}
```

Important!

You can only pass **local variables** in this way.

Specify another template

If you don't want to use the default template, you can specify your own template file during the **render(...)** call. Just pass it as the render method's first parameter:

Example:

```
public static void show(Long id) {
    Client client = Client.findById(id);
    render("Clients/showClient.html", id, client);
}
```

Redirect to another URL

The **redirect(…)** method emits a Redirect event that in turn generates an HTTP Redirect response.

```
public static void index() {
    redirect("http://www.zenexity.fr");
}
```

Action chaining

There is no equivalent to the Servlet API **forward**. An HTTP request can only invoke one action. If you need to invoke another action, you have to redirect the browser to the URL able to invoke that action. In this way, the browser URL is always consistent with the executed action, and the **Back/Forward/Refresh** management is much easier.

You can send a Redirect response to any action, simply by invoking the action method in a Java way. The Java call is intercepted by the framework and the correct HTTP Redirect is generated.

For example:

```
public class Clients extends Controller {

    public static void show(Long id) {
        Client client = Client.findById(id);
        render(client);
    }

    public static void create(String name) {
        Client client = new Client(name);
        client.save();
        show(client.id);
    }

}
```

With these routes:

```
GET    /clients/{id}           Clients.show
POST   /clients                Clients.create
```

- The browser sends a POST to the /clients URL.
- The Router invokes the Clients controller's create action.
- The action method calls the show action method directly.
- The Java call is intercepted and the Router reverse route generation creates the URL needed to invoke Clients.show with an id parameter.
- The HTTP Response is 302 Location:/clients/3132.
- The browser then issues GET /clients/3132.
- …

Interceptions

A controller can define interception methods. Interceptors are invoked for all actions of the controller class and its descendants. It's a useful way to define treatments that are common to all actions: verifying that a user is authenticated, loading request-scope information…

These methods have to be **static** but not **public**. You have to annotate these methods with a valid interception marker.

@Before

Methods annotated with the **@Before** annotation are executed before each action call for this Controller.

So, to create a security check:

```
public class Admin extends Application {

    @Before
    static void checkAuthentification() {
        if(session.get("user") == null) login();
    }

    public static void index() {
        List<User> users = User.findAll();
        render(users);
    }

    ...

}
```

If you don't want the @Before method to intercept all action calls, you can specify a list of actions to exclude:

```
public class Admin extends Application {

    @Before(unless="login")
    static void checkAuthentification() {
        if(session.get("user") == null) login();
    }

    public static void index() {
        List<User> users = User.findAll();
        render(users);
    }

    ...

}
```

Or if you want the @Before method to intercept a list of action calls, you can specify a only param :

```
public class Admin extends Application {

    @Before(only={"login","logout"})
    static void doSomething() {
                ...
        }

        ...
}
```

unless and only params are available for @After @Before and @Finally

@After

Methods annotated with the **@After** annotation are executed after each action call for this Controller.

```
public class Admin extends Application {

    @After
    static void log() {
        Logger.info("Action executed ...");
    }

    public static void index() {
        List<User> users = User.findAll();
        render(users);
    }

    ...

}
```

@Finally

Methods annotated with the **@Finally** annotation are executed after each action result is applied from for this Controller.

```
public class Admin extends Application {

    @Finally
    static void log() {
        Logger.info("Response contains : " + response.out);
    }

    public static void index() {
        List<User> users = User.findAll();
        render(users);
    }

    ...

}
```

Controller hierarchy

If a Controller class is a subclass of another Controller class, interceptions are applied to the full Controller hierarchy.

Adding more interceptors using the @With annotation

Because Java does not allow multiple inheritance, it can be very limiting to rely on the Controller hierarchy to apply interceptors. But you can define some interceptors in a totally different class, and link them with any controller using the **@With** annotation.

Example:

```
public class Secure extends Controller {

    @Before
    static void checkAuthenticated() {
        if(!session.containsKey("user")) {
            unAuthorized();
        }
    }
}
```

And on another Controller:

```
@With(Secure.class)
public class Admin extends Application {

    ...

}
```

Session and Flash scope

If you have to keep data across multiple HTTP Requests, you can save them in the Session or the Flash scope. Data stored in the Session are available during the whole user session, and data stored in the flash scope are available to the next request only.

It's important to understand that Session and Flash data are not stored in the server but are added to each subsequent HTTP Request, using the Cookie mechanism. So the data size is very limited (up to 4 KB) and you can only store String values.

Of course, cookies are signed with a secret key so the client can't modify the cookie data (or it will be invalidated). The Play session is not aimed to be used as a cache. If you need to cache some data related to a specific session, you can use the Play built-in cache mechanism and use the **session.getId()** key to keep them related to a specific user session.

Example:

```
public static void index() {
    List messages = Cache.get(session.getId() + "-messages", List.class);
    if(messages == null) {
        // Cache miss
        messages = Message.findByUser(session.get("user"));
        Cache.set(session.getId() + "-messages", messages, "30mn");
    }
    render(messages);
}
```

The cache has different semantics than the classic Servlet HTTP session object. You can't assume that these objects will be always in the cache. So it forces you to handle the cache miss cases, and keeps your application fully stateless.

Continuing the discussion

The next important layer of the MVC model is the View layer, for which Play provides an efficient templating system.

Play has an efficient templating system which allows to dynamically generate HTML, XML, JSON or any text-based formatted document. The template engine uses Groovy as an expression language. A tag system allows you to create reusable functions.

Templates are stored in the **app/views** directory.

Template syntax

A template file is a text file, some parts of which have placeholders for dynamically generated content. The template's dynamic elements are written using the Groovy language. Groovy's syntax is very close to Java's.

Dynamic elements are resolved during template execution. The rendered result is then sent as part of the HTTP response.

Expressions: ${…}

The simplest way to make a dynamic element is to declare an expression. The syntax used here is **${…}** . The result of evaluating the expression is inserted in place of the expression.

For example:

```
<h1>Client ${client.name}</h1>
```

If you can't be sure of client being null, there is a Groovy shortcut:

```
<h1>Client ${client?.name}</h1>
```

Which will only display the client name if the client is not null.

Template decorators : #{extends /} and #{doLayout /}

Decorators provide a clean solution to share a page layout (or design) across several templates.

> Use #{get} and #{set} tags to share variables between the template and the decorator.

Embedding a page in a decorator is the matter of a one liner:

```
#{extends 'simpledesign.html' /}

#{set title:'A decorated page' /}
This content will be decorated.
```

The decorator : **simpledesign.html**

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>#{get 'title' /}</title>
  <link rel="stylesheet" type="text/css" href="@{'/public/stylesheets/main.css'}"
/>
</head>
<body>
  <h1>#{get 'title' /}</h1>
  #{doLayout /}
  <div class="footer">Built with the play! framework</div>
</body>
</html>
```

Tags: #{tagName /}

A tag is a template fragment that can be called with parameters. If the tag has only one parameter, by convention it is called "arg" and its name can be omitted.

For example, this tag inserts a SCRIPT tag to load a JavaScript file:

```
#{script 'jquery.js' /}
```

A tag has to be closed, either directly or by an end tag:

```
#{script 'jquery.js' /}
```

or

```
#{script 'jquery.js'}#{/script}
```

For example the **list** tag allows iteration over any collection. It takes two mandatory parameters:

```
<h1>Client ${client.name}</h1>
<ul>
    #{list items:client.accounts, as:'account' }
        <li>${account}</li>
    #{/list}
</ul>
```

All dynamic expressions are escaped by the template engine to avoid XSS security issues in your application. So the **title** variable containing **<h1>Title</h1>** is now escaped:

```
${title} --> <h1>Title</h1>
```

If you really want to display it in an unescaped way, you need to explicitely call the **raw()** method:

```
${title.raw()} --> <h1>Title</h1>
```

Also, if you want to display a large part of raw HTML, you can use the **#{verbatim /}** tag:

```
#{verbatim}
    ${title} --> <h1>Title</h1>
#{/verbatim}
```

Actions: @{…} or @@{…}

You can use the Router to (reverse) generate a URL corresponding to a specified route. From a template you can use the special **@{…}** syntax to do that.

For example:

```
<h1>Client ${client.name}</h1>
<p>
   <a href="@{Clients.showAccounts(client.id)}">All accounts</a>
</p>
<hr />
<a href="@{Clients.index()}">Back</a>
```

The **@@{…}** syntax does the same but generates an absolute URL (notably useful for e-mail, …).

Messages: &{…}

If your application needs internationalization you can use the **&{…}** syntax to display an internationalized message:

For example in the files **conf/messages** we specify:

```
clientName=The client name is %s
```

To display this message in a template, simply use:

```
<h1>&{'clientName', client.name}</h1>
```

Comment: *{…}*

Comments aren't evaluated by the template engine. They are just comments…

```
*{**** Display the user name ****}*
<div class="name">
    ${user.name}
</div>
```

Scripts: %{…}%

A script is a more complicated set of expressions. A script can declare some variables and define some statements. Use the **{…}** syntax to insert a script.

```
%{
   fullName = client.name.toUpperCase()+' '+client.forname;
}%

<h1>Client ${fullName}</h1>
```

A script can write dynamic content directly using the **out** object:

```
%{
   fullName = client.name.toUpperCase()+' '+client.forname;
   out.print('<h1>'+fullName+'</h1>');
}%
```

You can use a script to create a structure such as an iteration in your template:

```
<h1>Client ${client.name}</h1>
<ul>
%{
     for(account in client.accounts) {
}%
     <li>${account}</li>
%{
     }
}%
</ul>
```

Bear in mind that a template is not a place to do complex things. So, use a tag when you can, or move the computations into the controller or the model object.

Template inheritance

A template can inherit another template, i.e. it can be included as a part of an other template.

To inherit another template, use the **extends** tag:

```
#{extends 'main.html' /}

<h1>Some code</h1>
```

The **main.html** template is a standard template, but it uses the **doLayout** tag to include the content:

```
<h1>Main template</h1>

<div id="content">
    #{doLayout /}
</div>
```

Create tags

You can easily create specific tags for your application. A tag is a simple template file, stored in the **app/views/tags** directory. The template's file name is used as the tag name.

To create a **hello** tag, just create the **app/views/tags/hello.html** file.

```
Hello from tag!
```

No need to configure anything. You can use the tag directly:

```
#{hello /}
```

Retrieve tags parameters

Tag parameters are exposed as template variables. The variable names are constructed with the '_' character prepended to the parameter name.

For example:

```
Hello ${_name} !
```

And you can pass the name parameter to the tag:

```
#{hello name:'Bob' /}
```

If your tag has only one parameter, you can use the fact than the default parameter name is **arg** and that its name is implicit.

Example:

```
Hello ${_arg}!
```

And you can call it easily using:

```
#{hello 'Bob' /}
```

Invoke tag body

If your tag supports a **body**, you can include it at any point in the tag code, using the **doBody** tag.

For example:

```
Hello #{doBody /}!
```

And you can then pass the name as tag body:

```
#{hello}
   Bob
#{/hello}
```

Java object extensions in templates

When you use your Java object within the template engine, new methods are added to it. These methods don't exist in the original Java class and are dynamically added by the template engine.

For example, to allow easy number formatting in a template, a **format** method is added to **java.lang.Number**.

It's then very easy to format a number:

```
<ul>
#{list items:products, as:'product'}
    <li>${product.name}. Price: ${product.price.format('## ###,00')}
€</li>
#{/list}
</ul>
```

<<<<<<< TREE
The same applies to **java.util.Date**.
===
You can find a list of available methods within the API documentation.
>>>>>>> MERGE-SOURCE

Create custom extensions

Your project may have specific formatting needs, in which case you can provide your own extensions.

You only need to create a Java class extending **play.templates.JavaExtensions**.

For instance, to provide a custom currency formatter for a number:

```
package ext;

import play.templates.JavaExtensions;

public class CurrencyExtensions extends JavaExtensions {

  public static String ccyAmount(Number number, String currencySymbol) {
     String format = "'"+currencySymbol + "'#####.##";
     return new DecimalFormat(format).format(number);
  }

}
```

Each extension method is a static method and should return a **java.lang.String** to be written back in the page. The first parameter will hold the enhanced object.

Use your formatter like this:

```
<em>Price: ${123456.324234.ccyAmount()}</em>
```

Template extension classes are automatically detected by Play at start-up. You just have to restart your application to make them available.

Implicit objects available from a template

All objects added to the **renderArgs** scope are directly injected as template variables.

For instance, to inject a 'user' bean into the template from a controller:

```
renderArgs.put("user", user );
```

When you render a template from an action, the framework also adds implicit objects:

- session, the Session object
- flash, the Flash scope
- request, the current Request

- params, HTTP params
- play, a reference to play.Play
- lang, the current language
- messages, the messages map
- out, the Writer

Continuing the discussion

The last layer of a Play application is the domain object model.

The model has a central position in a Play application. It's the domain-specific representation of the information on which the application operates.

Martin Fowler defines it as:

> Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

A common Java anti-pattern is to keep the model as a set of simple Java Beans and put the application logic back into a "service" layer which operates on the model objects.

Martin fowler again has named this anti-pattern Anemic object model:

> The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters. Indeed often these models come with design rules that say that you are not to put any domain logic in the domain objects. Instead there are a set of service objects which capture all the domain logic. These services live on top of the domain model and use the domain model for data.
>
> The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design, which is to combine data and process together. The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk. What's worse, many people think that anemic objects are real objects, and thus completely miss the point of what object-oriented design is all about.

Properties simulation

If you take a look at Play sample applications, you will often see that classes declare public variables. Now if you're a Java developer with any experience at all, warning sirens are probably clanging like mad at the sight of a public variable. In Java (as in other object-oriented languages), best practice says to make all fields private and provide accessors and mutators. This is to promote encapsulation, a concept critical to object oriented design.

Java has no truly built-in property definition system. It uses a convention named Java Beans: a property on a Java object is defined by a couple of getXxx/setXxx methods. If the property is read-only there is only a getter.

Although the system works well, it's very tedious to write. For each property you have to declare a private variable and write two methods. Thus, most of time the getter and setter implementation is always the same:

```
private String name;

public String getName() {
    return name;
}

public void setName(String value) {
    name = value;
}
```

The Model portion of the Play framework automatically generates this pattern while keeping your code clean. Effectively, all public variables become instance properties. The convention is that any **public**, **non static**, **non final** field of a class is seen as a property.

For example, when you define a class like this:

```
public class Product {

    public String name;
    public Integer price;

}
```

The loaded class will be:

```
public class Product {

    public String name;
    public Integer price;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getPrice() {
        return price;
    }

    public void setPrice(Integer name) {
        this.price = price;
    }

}
```

Then when you want to access a property you can just write:

```
product.name = "My product";
product.price = 58;
```

Which is translated at load time to:

```
product.setName("My product");
product.setPrice(58);
```

Warning!

You can't directly use getter and setter methods to access properties if you rely on automatic generation. These methods are generated at runtime. So if you reference them in code you write, the compiler won't find the methods and will generate an error.

Of course you can define the getter and setter methods yourself. If a method exists Play will use the existing accessors.

So to protect the Product class' price property value, you can write:

```
public class Product {

    public String name;
    public Integer price;

    public void setPrice(Integer price) {
        if (price < 0) {
            throw new IllegalArgumentException("Price can't be negative!");
        }
        this.price = price;
    }

}
```

Then if you try to set a property to a negative value an exception will be thrown:

```
product.price = -10: // Oops! IllegalArgumentException
```

Play will always use the defined getter or setter if it exists. Look at this code:

```
@Entity
public class Data extends Model {

   @Required
   public String value;
   public Integer anotherValue;

   public Integer getAnotherValue() {
       if(anotherValue == null) {
           return 0;
       }
       return anotherValue;
   }

   public void setAnotherValue(Integer value) {
       if(value == null) {
           this.anotherValue = null;
       } else {
           this.anotherValue = value * 2;
       }
   }

   public String toString() {
       return value + " - " + anotherValue;
   }

}
```

From another class, you can try these assertions:

```
Data data = new Data();
data.anotherValue = null;
assert data.anotherValue == 0;
data.anotherValue = 4
assert data.anotherValue == 8;
```

Yes it works. And because the enhanced class follows the Java Beans convention, when you use your object with a library that expects a JavaBean it will work perfectly!

Setup a database to persist your model objects

Most of the time you will need to save the model object data permanently. The most natural way is to save this data into a database.

During development, you can quickly set up an embedded database either in-memory or save it in a sub-directory within your application.

To start an in-memory HSQLDB database instance, just add this line to your **conf/application.conf** file:

```
db=mem
```

If your prefer to save the database in the file system, use:

```
db=fs
```

If you want to connect to an existing **MySQL5 server**, use the following:

```
db=mysql:user:pwd@database_name
```

You can also connect to any JDBC compliant database. Just add the driver library to your **lib/** directory and define JDBC properties in the application.conf:

```
db.url=jdbc:mysql://localhost/test
db.driver=com.mysql.jdbc.Driver
db.user=root
db.pass=
```

You can also specify a JPA dialect with the configuration option:

```
jpa.dialect=<dialect>
```

From your code, you can then obtain a **java.sql.Connection** from the **play.db.DB** and use it in the standard way.

```
Connection conn = DB.getConnection();
conn.createStatement().execute("select * from products");
```

Persist your object model with Hibernate

You can use Hibernate (through JPA) to persist your Java objects in the Database automatically.

When you define JPA entities by adding **@javax.persistence.Entity** annotations to any Java object, Play will automatically start a JPA entity manager.

```
@Entity
public class Product {

    public String name;
    public Integer price;

}
```

Warning!

A common mistake is to use the hibernate @Entity annotation instead of the JPA one. Remember that Play uses Hibernate through the JPA API.

You can then obtain the EntityManager from the **play.db.jpa.JPA** object:

```
EntityManager em = JPA.em();
em.persist(product);
em.createQuery("from Product where price > 50").getResultList();
```

Play provides a nice support class to help you to deal with JPA. Just extend **play.db.jpa.Model**.

```
@Entity
public class Product extends Model {

    public String name;
    public Integer price;

}
```

And then manipulate the Product object using simple methods on the Product instances:

```
Product.find("price > ?", 50).fetch();
Product product = Product.findById(2L);
product.save();
product.delete();
```

Keep the model stateless

Play is designed to operate in a 'share nothing' architecture. The idea is to keep the application completely stateless. By doing this you will allow your application to run on as many server nodes as needed at the same time.

What are the common traps you should avoid to keep the model stateless? **Do not store any object on the Java heap for multiple requests**

When you want to keep data across multiple requests you have several choices:

1. If data is small and simple enough, store it into the flash or the session scope. However these scopes are limited to about 4 KB each, and allow only String data.

2. Save the data permanently into persistent storage (like a database). For example if you need to create an object with a "wizard" that spans multiple requests:

- Initialize and persist the object into the database at the first request.
- Save the newly-created object's ID into the flash scope.
- During successive requests, retrieve the object from the database using the object ID, update it, and save it again.

3. Save the data temporarily into a transient storage (such as the Cache). For example if you need to create an object with a "wizard" that spans multiple requests:

- Initialize the object and save it into the Cache at the first request.
- Save the newly-created object's key into the flash scope
- During successive requests, retrieve the object from the cache (with the correct key), update it, and save it into the cache again.
- At the end of the last request in the chain, save the object permanently (into the database for example)

The Cache is not a reliable storage but if you put an object in the cache you should be able to retrieve it. Depending on your requirements, the Cache can be a very good choice and a good replacement for the Java Servlet session.

Continuing the discussion

Now we'll check how to perform operations outside any HTTP request using the jobs system.

Because Play is a web application framework, most of the application logic is done by controllers responding to HTTP requests.

But sometimes you will need to execute some application logic outside of any HTTP request. It can be useful for initialization tasks, maintenance tasks or to run long tasks without blocking the HTTP request execution pool.

Jobs are fully managed by the framework. That means that Play will manage all the database connection stuff, JPA entity manager synchronization and transactions management for you. To create a job you just need to extend the **play.jobs.Job** super class.

```
package jobs;

import play.jobs.*;

public class MyJob extends Job {

    public void doJob() {
        // execute some application logic here ...
    }

}
```

Sometimes you need to create jobs that return a result. You then override the **doJobWithResult()** method.

```
package jobs;

import play.jobs.*;

public class MyJob extends Job<String> {

    public String doJobWithResult() {
        // execute some application logic here ...
        return result;
    }

}
```

Here the use of **String** is just for the example, of course a job can return any object type.

Bootstrap job

Bootstrap jobs are executed by Play at application start time. To mark your job as a bootstrap job you just need to add the **@OnApplicationStart** annotation.

```
import play.jobs.*;

@OnApplicationStart
public class Bootstrap extends Job {

    public void doJob() {
        if(Page.count() == 0) {
            new Page("root").save();
            Logger.info("The page tree was empty. A root page has been created.");
        }
    }

}
```

You don't need to return a result. Even if you do it, the result will be lost.

Warning

When you run the application in DEV mode, the application waits for the first HTTP request to start. Moreover when you are in DEV mode, the application will sometimes automatically restart when needed.

When you run in PROD mode, the application will start synchronously with the server start.

Scheduled job

Scheduled jobs are run periodically by the framework. You can ask Play to run a job at a specific interval using the **@Every** annotation.

```
import play.jobs.*;

@Every("1h")
public class Bootstrap extends Job {

    public void doJob() {
        List<User> newUsers = User.find("newAccount = true").fetch();
        for(User user : newUsers) {
            Notifier.sayWelcome(user);
        }
    }

}
```

If the **@Every** annotation is not enough you can use the **@On** annotation to run your jobs using a CRON expression.

```
import play.jobs.*;

/** Fire at 12pm (noon) every day **/
@On("0 0 12 * * ?")
public class Bootstrap extends Job {

    public void doJob() {
        Logger.info("Maintenance job ...");
        ...
    }

}
```

Tip

We use the CRON expression parser from the Quartz library.

You don't need to return a result. Even if you do it, the result will be lost.

Suspendable requests

Play is intended to work with very short requests. It uses a fixed thread pool to process requests queued by the HTTP connector. To get optimum results, the thread pool should be as small as possible. We typically use the optimum value of **nb of processors + 1** to set the default pool size.

That means that if a request is very long (for example waiting for a long computation) it will block the thread pool and penalize your application responsiveness. Of course you could add more threads to the pool, but it will result in wasted resources, and anyway the pool will never be infinite.

Think for example about a chat application where browsers send a blocking HTTP request that waits for a new message to display. These requests can be very very long (typically several seconds) and will block the thread pool. If you plan to allow 100 users to connect simultaneously to your chat application you will need to provision at least 100 threads. OK it's feasible. But what about 1,000 users? Or 10,000?

To resolve these use cases, Play allows you to temporarily suspend a request. The HTTP request will stay connected, but the request execution will be popped out of the thread pool and tried again later. You can either tell Play to try the request later after a fixed delay, or wait for a Job completion.

Tip

You can take a look at the **chat** sample application to see a real example.

For example, this action will launch a very long job and wait for its completion before returning the result to the HTTP response:

```
public static void generatePDF(Long reportId) {
    if(request.isNew) {
        Report report = Report.findById(reportId);
        Future<InputStream> task = new ReportAsPDFJob(report).now();
        request.args.put("task", task),
        waitFor(task);
    }
    renderBinary((Future<InputStream>)request.args.get("task").get());
}
```

This action will be called twice. On the first call the **request.isNew** tag is set to **true**, so the action will create a Job and start it immediately. Then it tells Play to wait for the task completion before trying the action again. After the task is completed, Play will call the action a second time. This time the **request.isNew** tag is set to false. The action will retrieve the job result and send it as the HTTP response.

Continuing the discussion

Finally, let's see how you can test your applications.

Creating automatic test suites for your application is a good way to make it robust. It allows you to work in a very agile way.

Play tests are built using JUnit 4 or Selenium depending what you want to test.

Writing tests

The tests must be created in the **test/** directory. This folder will only be added to the sources path when the application is run in **test** mode. You can write 3 different kinds of tests.

Unit test

A unit test is written using JUnit. In this kind of test you can test the model of your application (including some utilities).

Here is an example of a Unit test:

```
import play.test.*;
import org.junit.*;

public class MyTest extends UnitTest {

    @Test
    public void aTest() {
        assertEquals(2, 1 + 1); // A really important thing to test
    }

    @Test
    public void testUsers() {
        assertEquals(3, Users.count());
    }

}
```

Functional test

A functional test is written using JUnit. In this kind of test you can test your application by accessing directly the controller objects.

Here is an example of a Functional test:

```
import play.test.*;
import play.mvc.*;
import play.mvc.Http.*;
import org.junit.*;

public class ApplicationTest extends FunctionalTest {

    @Test
    public void testTheHomePage() {
        Response response = GET("/");
        assertStatus(200, response);
    }

}
```

Selenium test

Acceptance tests are written using Selenium. Here you can test your application by running it in an automated browser.

Selenium tests are written using HTML tables. You can either use this native syntax or use the **#{selenium /}** tag.

Here is an example of a Selenium test:

```
#{selenium 'Test security'}

    // Try to log in the administration area
    clearSession()
    open('/admin')
    assertTextPresent('Login')
    type('login', 'admin')
    type('password', 'secret')
    clickAndWait('signin')

    // Verify that the user in correctly logged in
    assertText('success', 'Welcom admin!')

#{/selenium}
```

Fixtures

When you run tests, you need to have stable data for your application. The simplest way is to reset your database before each test.

The **play.test.Fixtures** class helps you to manipulate your database and to inject test data. You typically use it in a @Before method of a JUnit test.

```
@Before
public void setUp() {
    Fixtures.deleteAll();
}
```

To import data, it is simpler to define them in a YAML file that the Fixtures helper can automatically import.

```
# Test data

Company(google):
   name:    Google

Company(zen):
   name:    Zenexity

User(guillaume):
   name:    guillaume
   company: zen
```

And then:

```
@Before
public void setUp() {
    Fixtures.deleteAll();
    Fixtures.load("data.yml");
}
```

For Selenium tests, you can use the **#{fixture /}** tag:

```
#{fixture delete:'all', load:'data.yml' /}

#{selenium}

    // Write your test here

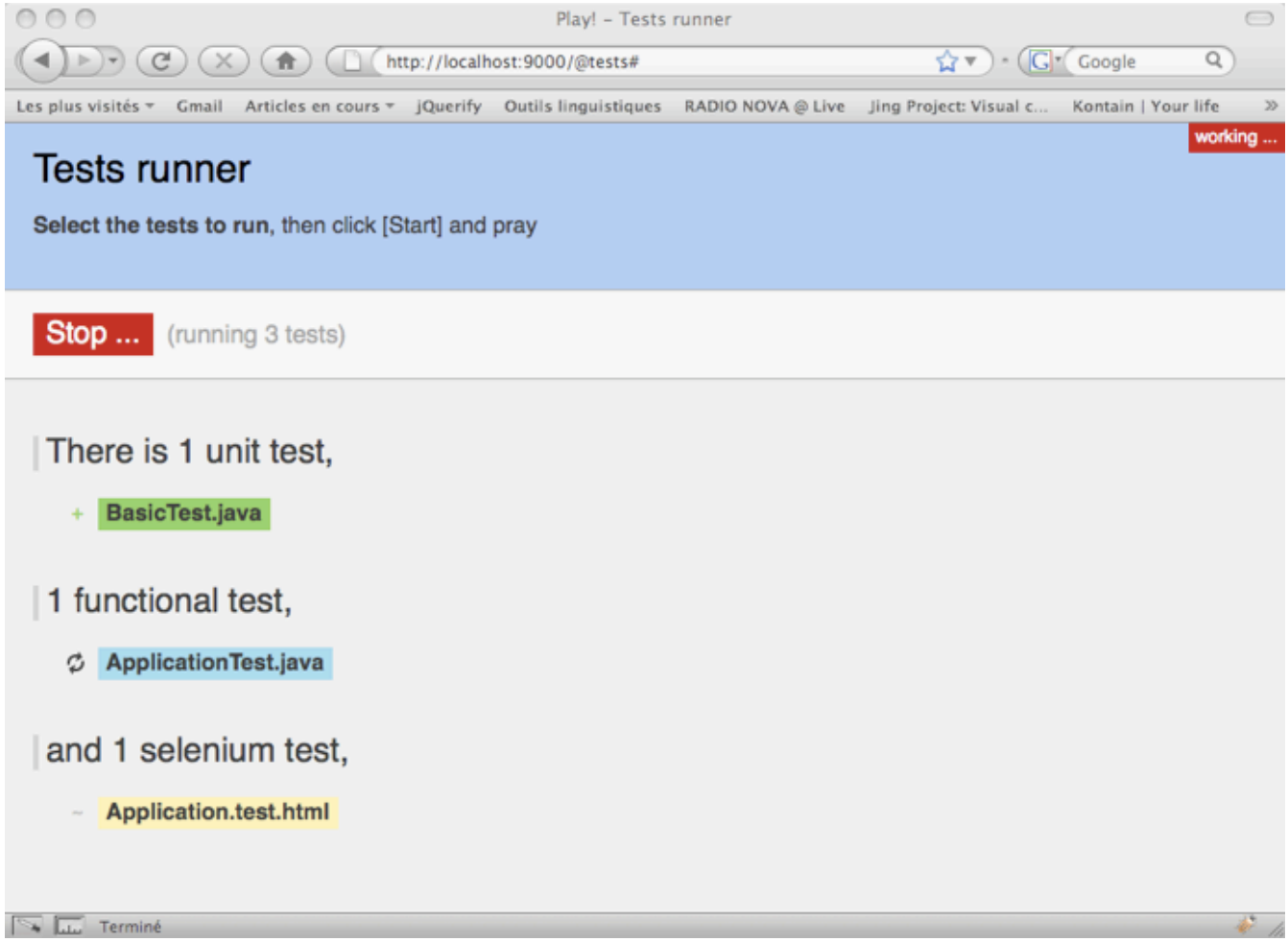
#{/selenium}
```

Running the tests

To run the tests, you must run your application in **test** mode using the **play test** command.

```
# play test myApp
```

In this mode, Play will automatically load the **test-runner** module. This module provides a Web based test runner, available at the http://localhost:9000/@tests URL.



When you run a test, the result is saved into the **/test-result** directory of your application.

On the test runner page, each test is a link. You can 'right click' and 'Open in a new tab', to run the test directly outside of the test-runner.

When you run tests this way, Play will start with a special **test** framework ID. So you can define special configurations in the **application.conf** file.

For example:

```
%test.db=mem
%test.jpa.ddl=create-drop
```

Continuous integration, and running the tests automatically

The **auto-test** command does the same as the **test** command, but it automatically launches a browser, runs all the tests, and stops.

This is a useful command if you want to set up a continuous integration system;

After the run, all results are saved to the **/test-result** directory. Moreover, this directory contains a marker file (either **result.failed** or **result.passed**) for the test suite's final result. Finally, this directory contains all the logs, in an **application.log** file.

So setting up a continuous integration system to test your application, could be:

- Checkout the latest version of your application
- Run play auto-test
- Wait for the end of the process
- Check for the marker file result.passed or result.failed in the /test-result directory

Run these steps in a CRON tab, and you're done!

Play applications can be deployed virtually anywhere: inside Servlet containers, as standalone servers, in Google Application Engine, Stack, a Cloud, etc...

Standalone Play application

The simplest and the more robust way is to simply run your Play application without any container. You can use a frontal HTTP server like Lighttpd or Apache if you need more advanced HTTP features like virtual hosting.

The built-in HTTP server can serve thousands of HTTP requests per second so it will never be the performance bottleneck. Moreover it uses a more efficient threading model (where a Servlet container uses 1 thread per request). Different modules allow you to use different servers (Grizzly, Netty, etc...) as well.

Those servers support long polling and allow to manage very long requests (waiting for a long task to complete), and direct streaming of File objects (and any InputStream if you specify the Content-Length), without blocking the execution thread.

You will have less problems running your application this way, as you will use the same environment that you used during the development process. A lot of bugs can be discovered only when you deploy to a JEE application server (different home dir, classloader issues, library conflicts, etc...).

Please refer to the 'Put your application in production' page for more information.

Application servers

Your Play application can also run inside your favorite application server. Most application servers are supported out of the box.

Application server compatibility matrix

JBoss 4.2.x	JBoss 5.x	JBoss 6M2	Glasshfish v3	IBM Websphere 6.1	IBM Websphere 7	Geronimo 2.x	Tomcat 6.x	Jetty 7.x	Resin 4.0.5

These application server are known to work with Play 1.0.2; but feel free to report any other working deployment.

Deploying

You need to package your application as a war file. This is easily done with the following command:

```
play war myapp -o myapp.war
```

Please note that your application server must support deployment of exploded WAR files.

You are now ready to deploy your application.

You are advised to 'isolate' your Play application from the other applications to avoid version mismatches between the application libraries. This step is not standardized by the JEE / Servlet Container specification, and is therefore vendor specific.

We recommend you refer to your application server manual in order to 'isolate' your WAR. As an example below is how you isolate a war file in JBoss Application server. Note that this is an optional step:

Insert the following content (or create the file) in your application war directory at myapp.war/WEB-INF/jboss-web.xml:

```
<jboss-web>
 <class-loading java2classloadingcompliance="false">
 <loader-repository>
 com.example:archive=unique-archive-name
 <loader-repository-config>java2ParentDelegation=false</loader-repository-config>
</loader-repository>
</class-loading>
</jboss-web>
```

Replace com.example:archive=unique-archive-name with whatever you wish as long as it is unique.

Google Application Engine (GAE)

A Play application can very easily be deployed to the GAE. It is a matter of installing the relevant GAE module.

```
play install gae
```

Deploying to the Google Application Engine is again really easy:

```
play gae:deploy myapp
```

Refer to the module documentation for more information.

Stax cloud hosting platform

Easy deployment to the Stax cloud hosting platform: again nothing could be easier. Install the Stax module and deploy within seconds.

Refer to the module documentation for more information.

A Play application can be assembled from several application modules. This allows you to reuse application components across several applications or split a large application into several smaller applications.

What is a module?

A module is just another Play application; however some differences exist in the way resources are loaded for an application module:

- A module does not have a conf/application.conf file.
- A module can have a conf/routes file, but these routes will not be loaded automatically.
- All files are first searched for in the main application path, then in all loaded modules.
- A module can contain plain Java code packaged as a JAR file in the module/lib directory.
- Everything in a module is optional.

You can create a module with the **play new-module** command.

How to load a module from an application

The default modules installation path is in the **$PLAY_HOME/modules** directory. This way all modules are automatically available to all applications run with the framework.

If you want to load external modules from an application, just declare external modules in the application.conf file in the main application.

```
# Additional modules
# ~~~~~
# A module is another Play application. Add a line for each module you want
# to add to your application. Module paths are either absolute or relative to
# the application root.
#
module.crud=${play.path}/modules/crud
```

You can also create a standalone project that contains all modules needed by your application. You just need to copy all modules into the same directory as your application.

Let's say you have a large application with a CMS component, a forum component, a directory component and using the GWT module. You can create a project using this layout:

```
/my-project
    /cms
    /forum
    /directory
    /gwt
    /main
```

Here **main** is the directory of your main application (created using the **play new** command), **gwt** is a module installed from the modules repository, and **cms**, **forum** and **directory** are modules created using the **play new-module** command.

Now from the main application configuration file (my-project/main/conf/application.conf), you can load these modules using:

```
# Additional modules
# ~~~~~
mdoule.gwt=../gwt
module.cms=../cms
module.forum=../forum
module.directory=../directory
```

When you run the main application (using **play run my-project/main**) it will load all these modules in memory as a larger application.

> If module paths are relative, they are resolved from the main application root.

Load default routes from modules

A module can provide a default routes file. You can load it in the main application routes file, using a special route declaration:

```
# Import the default CRUD routes
GET     /admin       module:crud
```

You can even load routes from all available modules:

```
GET     /         module:*
```

Using the module repository

The module repository identifies all modules contributed by the community. A module can have several versions. You have to check the module's documentation for which version you need to use for your framework version.

You can also browse the module repository using the **play list-modules** command.

```
gbo-mac:~ guillaume$ play11 list-modules
~        _            _
~  _ __ | | __ _ _  _| |
~ | '_ \| |/ _' | || |_|
~ |  __/|_|\____|\__ (_)
~ |_|            |__/
~
~ play! 1.1-unstable-r761, http://www.playframework.org
~
~ You can also browse this list online at http://www.playframework.org/modules
~
~ [bespin]
~   Bespin online editor
~   http://www.playframework.org/modules/bespin
~   Versions: 1.0, 1.0.1
~
~ [cobertura]
~   Cobertura
~   http://www.playframework.org/modules/cobertura
~   Versions: 1.0
...
```

You can install a module using the **play install {module}-{version}** command. For example, to install the Scala support to the framework, use:

```
play install scala-head
```

By convention the **head** version is the unstable version of the module. You can also install the default version of a module by omitting the version information. For example:

```
play install scala
```

Modules installed this way are downloaded to the **/modules** directory of your framework installation. You can change the installation path using the **--path** option:

```
play install gwt --path=my-project
```

Contributing a new module to the module repository

First you need to have an OpenID. It will help us to authenticate you as author of your modules. Then send us a module registration request on the Google Group.

Please tell us:

- More about your module. What is it?
- Your module name. It must match the [a-zA-Z]+ regular expression.
- A short description of the module.
- Your project home page.
- Your OpenID.
- Your module must be hosted somewhere with the source code available and a way to report bugs. If you don't have any idea, github, Google Code and Launchpad are good choices.

To release your module, simply use the play build-module command. Then connect to the module repository and upload the generated package.

You can of course use the offical Google Group to provide help and share information about your work.

The Play framework is designed with security in mind - but it is impossible to prevent developers using the framework to mess up and open holes. The guide describes the common security issues in web applications and how to avoid them in Play applications.

Sessions

Often you need to keep information associated with a user, in particular the logged-in status. Without a session, the user would need to pass credentials with each request.

That's what sessions are for: a set of cookies stored in the user's browser that identify him or her to the web site, and provide other information your web application may choose to store there rather than in the data layer; for example the language.

Keep your secret… secret

The session is a hash of key/values, signed but not encrypted. That means that as long as your secret is safe, it is not possible for a third-party to forge sessions.

The secret is stored in conf/application.conf. It is very important to keep it private: do not commit it in a public repository, and when you install an application written by someone else change the secret key to your own. You can do that with the command "play secret".

Don't store critical data

However, since it is not encrypted, you should not store critical data in the session. It can be seen by looking at the user cookie, by sniffing the connection on a local network or over wifi.

The session is stored in a cookie, and cookies are limited to 4 KB. In addition to this limit, only strings can be stored.

Cross-Site Scripting

Cross-site scripting is one of the most common vulnerabilities in web applications. It consists of injecting malicious JavaScript into web pages using forms that your application provides.

Let's say you're writing a blog, and anyone can add a comment. If you blindly include what commenters have written into your HTML page, you're opening your site to attacks. It can be:

- Show a popup to your visitors
- Redirect your visitors to a site controlled by the attacker
- Steal information supposed to be visible only to the current user, and send it back to the attacker's site

Consequently it is very important to be protected from those attacks.

Since version 1.0.1, Play's template engine automatically escapes strings. For that, you need to ensure escapeInTemplates is enabled in your application.conf. It is by default in newly created applications. If your application was created prior to 1.0.1 release, it may not be enabled.

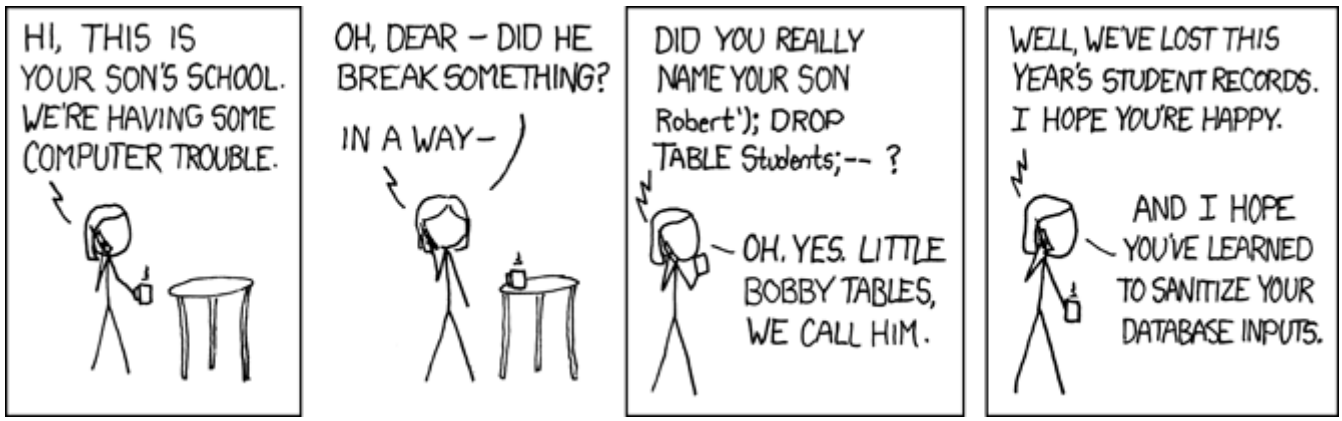
```
future.escapeInTemplates=true
```

It is strongly recommended to enable this option. When you really need to insert unescaped HTML in your templates, you can do so using the raw() method on the string. But if the string comes from a user input, you need to make sure it is sanitized first.

When sanitizing user inputs, always prefer whitelisting (only allow a list of safe tags) to blacklisting (forbid a list of unsafe tags and allow all the rest).

SQL Injection

SQL injection is an exploit consisting in using user input to execute a SQL query that was not intended by the developer. This can be used to destroy data, or to get access to data that shouldn't be visible to the current user.



When you are using high level "find" methods, you should be covered against SQL injection. When you build your own queries manually, you need to be careful not to concatenate strings with + but use "?" marks.

This is good:

```
createQuery("SELECT * from Stuff WHERE type= ?1").setParameter(1, theType);
```

This is NOT:

```
createQuery("SELECT * from Stuff WHERE type=" + theType;
```

Cross-Site Request Forgery

The CSRF attack can be really problematic in web applications:

> This attack method works by including malicious code or a link in a page that accesses a web application that the user is believed to have authenticated. If the session for that web application has not timed out, an attacker may execute unauthorized commands.

To prevent this attack, the first thing to do is to use GET and POST methods appropriately. That means that only the POST method should be used to change the application's state.

For POST requests the only way to secure critical actions properly is to issue an authenticity token. Play 1.0.2 now has a built-in helper to handle that:

- a new checkAuthenticity() method available in controllers, that checks for a valid authenticity token in the request parameters and sends a forbidden response if something is bad.
- session.getAuthenticityToken() generates an authenticity token only valid for the current session.
- #{authenticityToken /} creates a hidden input field that you can add to any form.

So for example:

```
public static destroyMyAccount() {
    checkAuthenticity();
    ...
}
```

Will only work when called from a form including a proper authenticity token:

```
#{form @ destroyMyAccount()}
    #{authenticityToken /}
    <input type="submit" value="destroy my account">
#{/form}
```

You can of course add this as a before filter if you want to protect all actions of a hierarchy of controllers.

More on cross-site request forgery

Play provides a set of very useful helpers to simplify the management of your JPA entities.

> **Note** that you can still go back to the plain JPA API whenever you want.

Starting the JPA entity manager

Play will automatically start the Hibernate entity manager when it finds one or more classes annotated with the **@javax.persistence.Entity** annotation. However, make sure that you have correctly configured a JDBC datasource or it will fail.

Obtaining the JPA entity manager

When the JPA entity manager is started you can get it from the application code, using the JPA helper. For example:

```
public static index() {
    Query query = JPA.em().createQuery("select * from Article");
    List<Article> articles = query.getResultList();
    render(articles);
}
```

Transaction management

Play will automatically manage transactions for you. It will start a transaction for each HTTP request and commit it when the HTTP response is sent. If your code throws an exception, the transaction will automatically rollback.

If you need to force transaction rollback from the application code, you can use the **JPA.setRollbackOnly()** method.

The play.db.jpa.Model support class

This is the main helper class for JPA. If you make one of your JPA entities extend the **play.db.jpa.Model** class, it will give you a lot of helper methods to simplify the JPA access.

For example, look at this Post model object:

```
@Entity
public class Post extends Model {

    public String title;
    public String content;
    public Date postDate;

    @ManyToOne
    public Author author;

    @OneToMany
    public List<Comment> comments;

}
```

The **play.db.jpa.Model** class automatically provides an **autogenerated Long id** field. We think that it's generally a good idea to keep an auto-generated Long id as primary key for JPA models (the technical primary key) and manage your functional primary key using another field.

Note that we have used the fact that Play automatically considers the Post class' **public** members as **properties**. So we don't need to write all setter/getter methods for this object.

Finding objects

The **play.db.jpa.Model** gives you several ways to find data. For example:

Find by ID

The simplest way to find an object.

```
Post aPost = Post.findById(5L);
```

Find all

```
List<Post> posts = Post.findAll();
```

This is the simplest way to retrieve **all** posts, but you can do the same using:

```
List<Post> posts = Post.all().fetch();
```

This allows you to paginate results:

```
List<Post> posts = Post.all().fetch(100); // 100 max posts
```

or even,

```
List<Post> posts = Post.all().from(50).fetch(100); // 100 max posts start at 50
```

Find using a simplified query

That allows you to create very expressive finders, but will only work for simple queries.

```
Post.find("byTitle", "My first post").fetch();
Post.find("byTitleLike", "%hello%").fetch();
Post.find("byAuthorIsNull").fetch();
Post.find("byTitleLikeAndAuthor", "%hello%", connectedUser).fetch();
```

Find using a JPQL query

You can use a JPQL query:

```
Post.find(
    "select p from Post p, Comment c where c.post = p and c.subject like ?", "%hop%"
);
```

or even a part of:

```
Post.find("title", "My first post").fetch();
Post.find("title like ?", "%hello%").fetch();
Post.find("author is null").fetch();
Post.find("title like % and author is null", "%hello%").fetch();
Post.find("title like % and author is null order by postDate", "%hello%").fetch();
```

You can even specify only the **order by** statement:

```
Post.find("order by postDate desc").fetch();
```

Counting objects

You can easily count objects.

```
long postCount = Post.count();
```

Or even count using a query:

```
long userPostCount = Post.count("author = ?", connectedUser);
```

Explicit save

Hibernate maintains a cache of Objects that have been queried from the database. These Objects are referred to as persistent Objects as long as the EntityManager that was used to fetch them is still active. That means that any changes to these Objects within the bounds of a transaction are automatically persisted when the transaction is committed. In standard JPA, these updates are implicit within the transaction's boundary; you don't have to explicitly call any method to persist the values.

The main downside is that we must manage all of our Objects manually. Instead of telling the EntityManager to update an Object (which would be far more intuitive), we must tell the EntityManager which Objects NOT to update. We do this by calling **refresh()**, which essentially rolls back a single entity. We do this just prior to calling commit on the transaction or when we realise the Object shouldn't be updated.

Here is a common use case, when editing a persistent object after a form submit:

```
public static void save(Long id) {
    User user = User.findById(id);
    user.edit(params);
    validation.valid(user);
    if(validation.hasErrors()) {
        // Here we have to explicitly discard the user modifications...
        user.refresh();
        edit(id);
    }
    show(id);
}
```

From what I've seen, most developers are not aware of this, and forget to discard the object state in case of errors, assuming that the object will not be saved without an explicit call to save().

So that's exactly what we've changed in Play. All the persistent objects extending the JPASupport/JPAModel will not be saved without an explicit call to the save() method. So you can actually rewrite the previous code as:

```
public static void save(Long id) {
    User user = User.findById(id);
    user.edit(params);
    validation.valid(user);
    if(validation.hasErrors()) {
        edit(id);
    }
    user.save();
    show(id);
}
```

This is far more intuitive. Moreover since it could be tedious to explicitly call save() on a large object graph, the save() call is automatically cascaded to the relationships annotated with the **cascade=CascadeType.ALL** attribute.

More about generic typing problems

The **play.db.jpa.Model** defines a set of generic methods. These generic methods use a type parameter to specify the method's return type. When using those methods, the concrete type to be used as return value is derived from the invocation context using type inference.

For example, the **findAll** method is defined as:

```
<T> List<T> findAll();
```

And you use it as:

```
List<Post> posts = Post.findAll();
```

Here the Java compiler resolves the actual type of **T** using the fact that you assign the method result to a List<Post>. So T is resolved as a Post type.

Unfortunately, this doesn't work if the generic method's return value is directly used as a parameter for another method invocation or used in a loop. So the following code fails with a compiler error saying "Type mismatch: cannot convert from element type Object to Post":

```
for(Post p : Post.findAll()) {
    p.delete();
}
```

Of course you can resolve this issue using a temporary local variable, as:

```
List<Post> posts = Post.findAll(); // type inference works here!
for(Post p : posts) {
    p.delete();
}
```

But wait, there is a better way. You can use a practical but somewhat unknown Java language feature, which makes the code shorter while more readable at the same time:

```
for(Post p : Post.<Post>findAll()) {
    p.delete();
}
```

Validations ensure that the data has certain values or meets specific requirements. You can use validation to verify that your models are correct before saving them to the database, or use them directly on HTTP parameters to validate a simple form.

How does it work?

Each request has it own **Validation** object which collects errors. From a controller, you access it directly using the **validation** variable. You can still access a subset of the API using the **play.data.validation.Validation** class' static methods.

The validation object maintains a collection of **play.data.validation.Error** objects. Each error has two properties:

- The key. This helps you to determine which data element caused the error. The key value can be set arbitrarily but when Play generates errors, it uses default conventions that follow the Java variables' names.
- The message. This contains the error's textual description. The message can be a plain message or refer to a key from a message bundle (typically for internationalization support).

Let's see how to validate a simple HTTP parameter:

```
public static void hello(String name) {
     validation.required(name);
     ...
}
```

This code checks that the name variable is correctly set. If not, the corresponding error is added to the current errors collection.

You can repeat this operation for each validation you need:

```
public static void hello(String name, Integer age) {
     validation.required(name);
     validation.required(age);
     validation.min(age, 0);
     ...
}
```

Retrieving error messages

At the end of the validation you can check if any errors have been created and display them:

```
public static void hello(String name, Integer age) {
     validation.required(name);
     validation.required(age);
     validation.min(age, 0);

     if(validation.hasErrors()) {
         for(Error error : validation.errors()) {
             System.out.println(error.message());
         }
     }
}
```

Assuming that name and age are null, this would display:

```
name is required
age is required
```

Default messages are keys that refer to the message bundle. So in the **conf/messages** file you will have:

```
validation.required=%s is required
```

You can change this default message and override it for each application language. The **%s** placeholder will be replaced by the error key. You can override using the **error.message(String field)** method.

For example:

```
Error error = validation.required(name).error;
if(error != null) {
    System.out.println(error.message("The name"));
}
```

You can also specify a different message for each check:

```
Error error = validation.required(name).message("Fill the name!").error;
if(error != null) {
    System.out.println(error.message());
}
```

Displaying errors in the template

In most cases you want to display the error messages in the view template. You can access them in the template using the **errors** object. Some tags help you to display the errors:

Let's see a sample:

```
public static void hello(String name, Integer age) {
     validation.required(name);
     validation.required(age);
     validation.min(age, 0);
     render(name, age);
}
```

and now the template:

```
#{ifErrors}

   <h1>Oops...</h1>

   #{errors}
       <li>${error}</li>
   #{/errors}

#{/ifErrors}
#{else}

   Hello ${name}, you are ${age}.

#{/else}
```

But in a real application you want to redisplay the original form. So you will have two actions: one to display the form and another one to handle the POST.

Of course the validation will occur in the second action and if some error occurs you will have to redirect to the first action. In this case you need a special trick to keep your errors during the redirect. Use the **validation.keep()** method. This will save the errors collection for the next action.

Let's see a real sample:

```
public class Application extends Controller {

   public static void index() {
      render();
   }

   public static void hello(String name, Integer age) {
      validation.required(name);
      validation.required(age);
      validation.min(age, 0);
      if(validation.hasErrors()) {
          params.flash(); // add http parameters to the flash scope
          validation.keep(); // keep the errors for the next request
          index();
      }
      render(name, age);
   }

}
```

And the **view/Application/index.html** template:

```
#{ifErrors}
   <h1>Oops...</h1>

   #{errors}
       <li>${error}</li>
   #{/errors}
#{/ifErrors}

#{form @Application.hello()}
   <div>
      Name: <input type="text" name="name" value="${flash.name}" />
   </div>
   <div>
      Age: <input type="text" name="age" value="${flash.age}" />
   </div>
   <div>
      <input type="submit" value="Say hello" />
   </div>
#{/form}
```

You can create a better user experience by displaying each error message next to the field that generated the error:

```
#{ifErrors}
   <h1>Oops...</h1>
#{/ifErrors}

#{form @Application.hello()}
   <div>
      Name: <input type="text" name="name" value="${flash.name}" />
      <span class="error">#{error 'name' /}</span>
   </div>
   <div>
      Age: <input type="text" name="age" value="${flash.age}" />
      <span class="error">#{error 'age' /}</span>
   </div>
   <div>
      <input type="submit" value="Say hello" />
   </div>
#{/form}
```

Using annotations

You can use annotations to do the same thing:

```
public static void hello(@Required String name, @Required @Min(0) Integer age) {
   if(validation.hasErrors()) {
       params.flash(); // add http parameters to the flash scope
       validation.keep(); // keep the errors for the next request
       index();
   }
   render(name, age);
}
```

Validating objects

Using annotations you can easily add constraints to your model objects. Let's rewrite the previous example using a User class.

First the **User** class:

```
package models;

public class User {

    @Required
    public String name;

    @Required
    @Min(0)
    public Integer age;

}
```

Then the modified **hello** action:

```
public static void hello(@Valid User user) {
   if(validation.hasErrors()) {
       params.flash(); // add http parameters to the flash scope
       validation.keep(); // keep the errors for the next request
       index();
   }
   render(name, age);
}
```

And finally the modified form:

```
#{ifErrors}
   <h1>Oops...</h1>
#{/ifErrors}

#{form @Application.hello()}
   <div>
      Name: <input type="text" name="user.name" value="${flash['user.name']}" />
      <span class="error">#{error 'user.name' /}</span>
   </div>
   <div>
      Age: <input type="text" name="user.age" value="${flash['user.age']}" />
      <span class="error">#{error 'user.age' /}</span>
   </div>
   <div>
      <input type="submit" value="Say hello" />
   </div>
#{/form}
```

Custom validation

Can't find the validator you need in the **play.data.validation** package? Write your own. You can use the generic **@CheckWith** annotation to bind your own **Check** implementation.

For example:

```
public class User {

    @Required
    @CheckWith(MyPasswordCheck.class)
    public String password;

    static class MyPasswordCheck extends Check {

        public abstract boolean isSatisfied(Object user, Object password) {
            return notMatchPreviousPasswords(password);
        }

    }
}
```

The Play logger is built on Log4j. Since most Java libraries use Log4j or a wrapper able to use Log4j as a backend, you can easily configure logging that is well-suited to your application.

Logging from your application

Play provides a default logger with the class **play.Logger**. This class uses Log4j to write messages and exceptions to a logger named "play".

Logging from your application is easy:

```
Logger.info("A log message");
Logger.error(ex, "Oops");
```

The **play.Logger** class' methods support easy formatting through the standard Java formatter syntax:

```
Logger.debug("The param was %s", param);
Logger.info("I want to log %s and %s and %s", a, b, c);
```

You can still use Log4j directly to create alternative loggers for specific needs:

```
org.apache.log4j.Logger.getLogger("another.logger");
```

Configure log levels

You can configure the **play** logger's log level. Just define this key in the **application.conf** file:

```
application.log=INFO
```

You can change this value without restarting the server. Note that this level only applies to messages generated by the application.

If you need to fully configure Log4j, create a **log4j.properties** file in the **conf/** directory. Since this directory is the first element of the classpath, this file will be the default used by all libraries.

The default Log4j configuration is the following:

```
log4j.rootLogger=ERROR, Console

log4j.logger.play=INFO

# Console
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d{ABSOLUTE} %-5p ~ %m%n
```

Copy this file and update it for your specifics needs!

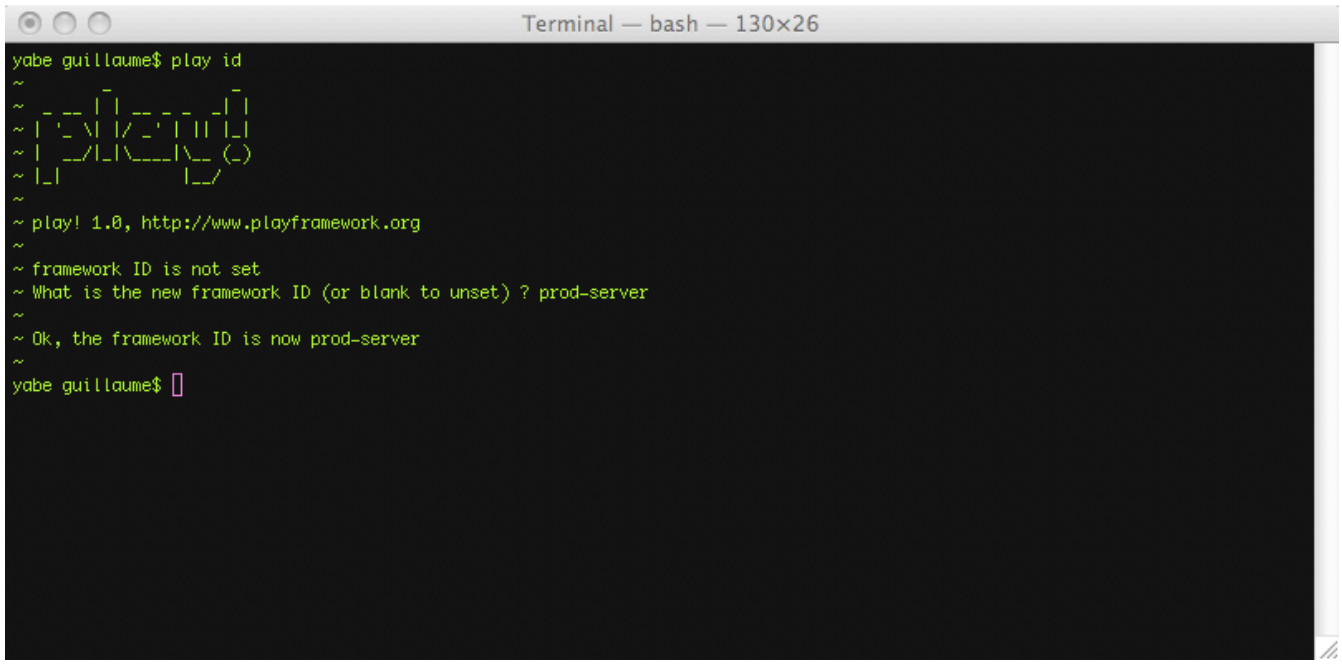
When you work in a team, different developers will use different configuration keys in their **application.conf**. For example, the log level, some database configuration… This generally leads to recurrent conflicts when you commit the file using your VCS.

Furthermore, different deployment environments - e.g. dev, test, staging and production - need different configurations.

The framework ID

To resolve this problem, Play allows you to give an **ID** to each framework installation. Use the play tool's **id** command to define this ID:

```
play id
```



You can then prefix your configuration keys with the framework ID for which the configuration option is intended:

```
application.name=Cool app
application.mode=dev
application.log=INFO

# Configuration for gbo
%gbo.application.log=DEBUG
%gbo.db=mem

# Configuration for src
%scr.http.port=9500

# Production configuration
%prod.http.port=80
%prod.application.log=INFO
%prod.application.mode=prod
```

Framework ID can be set from the command line

You can specify the framework ID to use for a particular command directly from the command line. For example to run an application in production mode you can use:

```
play run --%production
```

with these lines defined in the **application.conf** file:

```
application.mode=dev
%production.application.mode=prod
```

It should be compatible with all commands using the framework ID information. The default ID is still defined using the **play id** command.

By the way, **play test** is then equivalent to:

```
play run --%test
```

Internationalization is a means of adapting your application to different languages to allow for regional differences. Follow these steps to enable internationalization in your application.

Only use UTF-8!

Play supports only one encoding: UTF-8. Since encoding problems can be weird and difficult to deal with, we made the choice to support only one encoding. UTF-8 allows you to display all characters for all languages.

Be sure to be consistent with UTF-8 usage throughout your application:

- Edit all your source files as UTF-8
- Define proper encoding headers in HTTP
- Set your HTML meta tags to UTF-8
- If you use a database, configure it to use UTF-8, and always connect to it using UTF-8

Note

The UTF-8 encoding issue is the reason why most of the Play configuration files, even though they are Java properties files, are not named *.properties. Java imposes the requirement that properties files must be encoded with **iso-8859-1**. Play configuration files must be UTF-8 encoded. Need we say more?

Externalize your messages

To support i18n you have to externalize all messages in your application.

Create a file named **messages** in the application's **conf/** directory. This file is really just a Java properties file.

```
hello=Hello!
back=Back
```

Then you can define a specific **message** file for each language used in your application. Just add the ISO language code as a file extension.

For example, the message file containing the corresponding French translations is **conf/messages.fr**:

```
hello=Bonjour!
back=Retour
```

Define languages supported by the application

In the **conf/application.conf** file, define a list of supported languages:

```
application.langs=fr,en,ja
```

On the first request from a new user, Play will guess the default language to use. It does so by parsing the HTTP **Accept-language** header. It will then save the chosen language in a **PLAY_LANG** cookie. So the next request will use the same language.

From your application code your can retrieve the current language for the user by accessing the **play.i18n.Lang** object:

```
String lang = Lang.get();
```

If you want to permanently change the user language, use the change() method:

```
Lang.change("ja");
```

The new value will be saved back to the user's language cookie.

Define date format according to your locale

To set the default date format to use, edit your application.conf and set the following property:

```
date.format=yyy-MM-dd
    date.format.fr=dd/MM/yyyy
```

Please note that the language fr in the application.conf must be enabled as well (see above):
bc. application.langs=fr,en

This property affects how the dates are rendered in the templates using the ${date.format()}.
It also set the default date format when binding a date parameter.

Retrieve localized messages

From the application code, you can retrieve messages defined in message files. From Java, use the **play.i18n.Messages** object.

```
public static void hello() {
    renderText(Messages.get("hello"));
}
```

We support message formatting through the standard Java formatting syntax. You can also define dynamic content in your messages:

```
hello=Hello %s!
```

and

```
public static void hello(String user) {
    renderText(Messages.get("hello", user));
}
```

From a template you can use the special **&{...}** syntax to display localized messages:

```
<h1>&{'hello'}</h1>
```

or using dynamic content:

```
<h1>&{'hello', params.user}</h1>
```

To create high-performance systems, sometimes you need to cache data. Play has a cache library and will use memcached when used in a distributed environment.

If you don't configure memcached, Play will use a standalone cache that stores data in the JVM heap. Caching data in the JVM application breaks the "share nothing" assumption made by Play: you can't run your application on several servers, and expect the application to behave consistently. Each application instance will have a different copy of the data.

It is important to understand that the cache contract is clear: when you put data in a cache, you can't expect that data to remain there forever. In fact you shouldn't. A cache is fast, but values expire, and the cache generally exists only in memory (without persistent backup).

So the best way to use the cache is to repopulate it when it doesn't have what you expect:

```
public static void allProducts() {
    List<Product> products = Cache.get("products", List.class);
    if(products == null) {
        products = Product.findAll();
        Cache.set("products", products, "30mn");
    }
    render(products);
}
```

The cache API

The cache API is provided by the **play.cache.Cache** class. This class contains the set of methods to set, replace, and get data from the cache. Refer to the memcached documentation to understand the exact behavior of each method.

Some examples:

```
public static void showProduct(String id) {
    Product product = Cache.get("product_" + id, Product.class);
    if(product == null) {
        product = Product.findById(id);
        Cache.set("product_" + id, product, "30mn");
    }
    render(product);
}

public static void addProduct(String name, int price) {
    Product product = new Product(name, price);
    product.save();
    showProduct(id);
}

public static void editProduct(String id, String name, int price) {
    Product product = Product.findById(id);
    product.name = name;
    product.price = price;
    Cache.set("product_" + id, product, "30mn");
    showProduct(id);
}

public static void deleteProduct(String id) {
    Product product = Product.findById(id);
    product.delete();
    Cache.delete("product_" + id);
    allProducts();
}
```

Some methods start with the **safe** prefix - e.g. **safeDelete**, **safeSet**. The standard methods are non-blocking. That means that when you issue the call:

```
Cache.delete("product_" + id);
```

The **delete** method will return immediately and will not wait until the cached object is actually deleted. So if an error occurs - e.g. an IO error - the object may still be present.

When you need to make sure that the object is deleted before continuing, you can use the **safeDelete** method:

```
Cache.safeDelete("product_" + id);
```

This method is blocking and returns a boolean value indicating whether the object has been deleted or not. So the full pattern that ensures an item is deleted from the cache is:

```
if(!Cache.safeDelete("product_" + id)) {
    throw new Exception("Oops, the product has not been removed from the cache");
}
...
```

Note that those being blocking calls, **safe** methods will slow down your application. So use them only when needed.

Don't use the Session as a cache!

If you come from a framework that uses an in-memory Session implementation, you may be frustrated to see that Play allows only a small set of String data to be saved in the HTTP Session. But this is much better because a session is not the place to cache your application data!

So if you have been accustomed to doing things similar to:

```
httpServletRequest.getSession().put("userProducts", products);
...
// and then in subsequent requests
products = (List<Product>)httpServletRequest.getSession().get("userProducts");
```

In Play you achieve the same effect a little differently. We think it's a better approach:

```
Cache.put(session.getId(), products);
...
// and then in subsequent requests
List<Product> products = Cache.get(session.getId(), List.class)
```

Here we have used a unique UUID to keep unique information in the Cache for each user. Remember that, unlike a session object, the cache is not bound to any particular User!

Configure memcached

When you want to enable a real memcached implementation, enable memcached and define the daemon address in your **application.conf**:

```
memcached=enabled
memcached.host=127.0.0.1:11211
```

You can connect to a distributed cache by specifying multiple daemon addresses:

```
memcached=enabled
memcached.1.host=127.0.0.1:11211
memcached.2.host=127.0.0.1:11212
```

Here some simple tips to optimize your application for production.

Configure your application.conf for production

First off, the best way to specify production mode is to give a specific ID to your production framework. Let's pick **production** as an example. Refer manage application.conf in several environments to see how.

Set the framework in prod mode:

```
%production.application.mode=prod
```

In this mode, the framework will pre-compile all Java sources and templates. If an error is found at this step, the application will not start. Source modifications will not be hot reloaded.

Define a real database:

If you have used a development database (either **db=mem** or **db=fs**), you should configure a more robust database engine:

```
%production.db.url=jdbc:mysql://localhost/prod
%production.db.driver=com.mysql.jdbc.Driver
%production.db.user=root
%production.db.pass=1515312
```

Disable JPA automatic schema update:

If you have used the automatic schema update feature provided by Hibernate, you should disable this feature for production.

For your production server, it's usually a bad idea to let Hibernate automatically ALTER your production database's schema and data…

The initial deployment is potentially a different issue. In this case only specify:

```
%production.jpa.ddl=create
```

Define a secure secret key:

The Play secret key is used to secure cryptographic functions, like the session signature. Your application must keep this key very secret.

```
%production.application.secret=c12d1c59af499d20f4955d07255ed8ea333
```

You can use the **play secret** command to generate a new secure and random key (at least on a 'real' OS). If you plan to distribute your application to several servers, remember to use the **same key** for all application instances!

Configure logging

For production it's a good idea to use rolling log files. Do not send logging to the Console, since it will be written to the **logs/system.out** file and it will grow without bound!

Create a custom **log4j.properties** in the **conf/**, directory:

```
log4j.rootLogger=ERROR, Rolling

log4j.logger.play=INFO

# Rolling files
log4j.appender.Rolling=org.apache.log4j.RollingFileAppender
log4j.appender.Rolling.File=application.log
log4j.appender.Rolling.MaxFileSize=1MB
log4j.appender.Rolling.MaxBackupIndex=100
log4j.appender.Rolling.layout=org.apache.log4j.PatternLayout
log4j.appender.Rolling.layout.ConversionPattern=%d{ABSOLUTE} %-5p ~ %m%n
```

Set-up a front-end HTTP server

You can easily deploy your application as a stand-alone server by setting the application HTTP port to **80**:

```
%production.http.port=80
```

But if you plan to host several applications in the same server or load balance several instances of your application for scalability or fault tolerance, you can use a front-end HTTP server.

Note that using a front-end HTTP server will never give you better performance than using Play server directly!

Set-up with lighttpd

This example shows you how to configure lighttpd as a front-end web server. Note that you can do the same with Apache, but if you only need virtual hosting or load balancing, lighttpd is a very good choice and much easier to configure!

The **/etc/lighttpd/lighttpd.conf** file should define things like this:

```
server.modules = (
      "mod_access",
      "mod_proxy",
      "mod_accesslog"
)
...
$HTTP["host"] =~ "www.myapp.com" {
    proxy.balance = "round-robin" proxy.server = ( "/" =>
        ( ( "host" => "127.0.0.1", "port" => 9000 ) ) )
}

$HTTP["host"] =~ "www.loadbalancedapp.com" {
    proxy.balance = "round-robin" proxy.server = ( "/" => (
          ( "host" => "127.0.0.1", "port" => 9000 ),
          ( "host" => "127.0.0.1", "port" => 9001 ) )
    )
}
```

Set-up with Apache

The example below shows a simple set-up with Apache httpd server running in front of a standard Play configuration.

```
LoadModule proxy_module modules/mod_proxy.so
...
<VirtualHost *:80>
  ProxyPreserveHost On
  ServerName www.loadbalancedapp.com
  ProxyPass / http://127.0.0.1:9000/
  ProxyPassReverse / http://127.0.0.1:9000/
</VirtualHost>
```

Apache as a front proxy to allow transparent upgrade of your application

The basic idea is to run 2 Play instances of your web application and let the front-end proxy load-balance them. In case one is not available, it will forward all the requests to the available one.

Let's start the same Play application two times: one on port 9999 and one on port 9998.

Copy the application 2 times and edit the application.conf in the conf directory to change the port numbers.

For each web application directory:

```
play start mysuperwebapp
```

Now, let's configure our Apache web server to have a load balancer.

In Apache, I have the following configuration:

```
<VirtualHost mysuperwebapp.com:80>
  ServerName mysuperwebapp.com
  <Location /balancer-manager>
    SetHandler balancer-manager
    Order Deny,Allow
    Deny from all
    Allow from .mysuperwebapp.com
  </Location>
  <Proxy balancer://mycluster>
    BalancerMember http://localhost:9999
    BalancerMember http://localhost:9998 status=+H
  </Proxy>
  <Proxy *>
    Order Allow,Deny
    Allow From All
  </Proxy>
  ProxyPreserveHost On
  ProxyPass /balancer-manager !
  ProxyPass / balancer://mycluster/
  ProxyPassReverse / http://localhost:9999/
  ProxyPassReverse / http://localhost:9998/
</VirtualHost>
```

The important part is balancer://mycluster. This declares a load balancer. The +H option means that the second Play application is on stand-by. But you can also instruct it to load-balance.

Every time you want to upgrade mysuperwebapp, here is what you need to do:

```
play stop mysuperwebapp1
```

The load-balancer then forwards everything to mysuperwebapp2. In the meantime update mysuperwebapp1. Once you are done:

```
play start mysuperwebapp1
```

You can now safely update mysuperwebapp2.

Apache also provides a way to view the status of your cluster. Simply point your browser to /balancer-manager to view the current status of your clusters.

Because Play is completely stateless you don't have to manage sessions between the 2 clusters. You can actually easily scale to more than 2 Play instances.

Advanced proxy settings

When using an HTTP frontal server, request addresses are seen as coming from the HTTP server. In a usual set-up, where you both have the Play app and the proxy running on the same machine, the Play app will see the requests coming from 127.0.0.1.

Proxy servers can add a specific header to the request to tell the proxied application where the request came from. Most web servers will add an X-Forwarded-For header with the remote client IP address as first argument. If you enable the forward support in your Play app configuration:

```
XForwardedSupport=127.0.0.1,10.0.0.25
```

Play will change the request.remoteAddress from the proxy's IP to the client's IP. You have to list the IP addresses of your proxy servers for this to work.

However, the host header is untouched, it'll remain issued by the proxy. If you use Apache 2.x, you can add a directive like:

```
ProxyPreserveHost on
```

The host: header will be the original host request header issued by the client. By combining theses two techniques, your app will appear to be directly exposed.

Email are using the jakarta commons email library under the hoot. You can use the play.libs.Mail utility to send emails very easily:

A simple email:

```
SimpleEmail email = new SimpleEmail();
email.setFrom("sender@zenexity.fr");
email.addTo("recipient@zenexity.fr");
email.setSubject("subject");
email.setMsg("Message");
Mail.send(email);
```

An html email:

```
HtmlEmail email = new HtmlEmail();
email.addTo("info@lunatech.com");
email.setFrom(sender@lunatech.com", "Nicolas");
email.setSubject("Test email with inline image");
// embed the image and get the content id
URL url = new
URL("http://www.zenexity.fr/wp-content/themes/zenexity/file:///Users/nicolasleroux/Proje
cid = email.embed(url, "Zenexity logo");
// set the html message
email.setHtmlMsg("<html>Zenexity logo - <img
src=\"cid:"+cid+"\"></html>");
// set the alternative message
email.setTextMsg("Your email client does not support HTML messages, too bad :(");
```

More information about the commons email here

Mail and MVC integration

You can also send complex, dynamic e-mail using the standard templates mechanism and syntax.

First, define a **Mailer notifier** in your application. Your mailer notifier must subclass **play.mvc.Mailer** and be part of the **notifiers** package.

Each public static method will be an e-mail sender, in a similar manner as actions for an MVC controller.

For example:

```
package notifiers;

import play.*;
import play.mvc.*;
import java.util.*;

public class Mails extends Mailer {

   public static void welcome(User user) {
      setSubject("Welcome %s", user.name);
      addRecipient(user.email);
      addFrom("Me <me@me.com>");
      EmailAttachment attachment = new EmailAttachment();
      attachment.setDescription("A pdf document");
      attachment.setPath(Play.getFile("rules.pdf").getPath());
      addAttachment(attachment);
      send(user);
   }

   public static void lostPassword(User user) {
      String newpassword = user.password;
      addFrom("Robot <robot@thecompany.com>");
      setSubject("Your password has been reset");
      addRecipient(user.email);
      send(user, newpassword);
   }

}
```

text/html e-mail

The send call will render the app/views/Mails/welcome.html template as the e-mail message body.

```
<html><body><p>Welcome <b>${user.name}</b>, </p>
...
</html>
```

The template for the lostPassword method could look like this:

app/views/Mails/lostPassword.html

```
<html><body><head>...</head><body><img
src="mycompany.com/images"/><p>Hello ${user.name},<br/>
Your new password is <b>${newpassword}</b>.
</p>
</body>
</html>
```

text/plain e-mail

If no HTML template is defined, then a text/plain e-mail is sent using the text template.

The send call will render the app/views/Mails/welcome.txt template as the e-mail message body.

```
Welcome ${user.name},
...
```

The template for the lostPassword method could look like this:

app/views/Mails/lostPassword.txt

```
Hello ${user.name},

Your new password is ${newpassword}.
```

text/html e-mail with text/plain alternative

If an HTML template is defined and a text template exists, then the text template will be used as an alternative message. In our previous example, if both

app/views/Mails/lostPassword.html

and

app/views/Mails/lostPassword.txt

are defined, then the e-mail will be sent in text/html as defined in lostPassword.html with an alternative part as defined in lostPassword.txt. So you can send nice HMTL e-mail to your friends and still please those geeky friends that still use mutt ;)

SMTP configuration

First of all, you need to define the SMTP server to use:

```
mail.smtp.host=smtp.taldius.net
```

If your SMTP server requires authentication, use the following properties:

```
mail.smtp.user=jfp
mail.smtp.pass=topsecret
```

Channel & ports

There are two ways to send the e-mail over an encrypted channel. If your server supports the **starttls** command (see: RFC 2487), you can use a clear connection on port 25 that will switch to SSL/TLS. You can do so by adding this configuration option:

```
mail.smtp.channel=starttls
```

Your server may also provide a SMTP-over-SSL (SMTPS) connector, that is an SSL socket listening on port 465. In that case, you tell Play to use this setup using the configuration option:

```
mail.smtp.channel=ssl
```

More about configuration

Under the hood, Play uses JavaMail to perform the actual SMTP transactions. If you need to see what's going on, try:

```
mail.debug=true
```

When using SSL connections with JavaMail, the default SSL behavior is to drop the connection if the remote server certificate is not signed by a root certificate. This is the case in particular when using a self-signed certificate. Play's default behavior is to skip that check. You can control this using the following property:

```
mail.smtp.socketFactory.class
```

If you need to connect to servers using non-standard ports, the following property will override the defaults:

```
mail.smtp.port=2500
```

Using Gmail

To use Gmail's servers, use this configuration:

```
mail.smtp.host=smtp.gmail.com
mail.smtp.user=yourGmailLogin
mail.smtp.pass=yourGmailPassword
mail.smtp.channel=ssl
```

OpenID is an open and decentralized identity system. You can easily accept new users in your application without having to keep specific user information. You just have to keep track of authorized users through their **OpenID**.

A simple OpenID authentication example

This example provides a high-level view of how OpenID authentication can be used within a Play application:

- For each request, check if the user is connected
- If not, display a page where the user can submit his OpenID
- Redirect the user to the OpenID provider
- When the user comes back, get the verified OpenID and save it in the HTTP session.

The OpenID functionality is provided by the **play.libs.OpenID** class and is built using OpenID4Java.

```
@Before(unless={"login", "authenticate"})
static void checkAuthenticated() {
    if(!session.contains("user")) {
        login();
    }
}

public static void index() {
    render("Hello %s!", session.get("user"));
}

public static void login() {
    render();
}

public static void authenticate(String user) {
    if(OpenID.isAuthenticationResponse()) {
        UserInfo verifiedUser = OpenID.getVerifiedID();
        if(verifiedUser == null) {
            flash.error("Oops. Authentication has failed");
            login();
        }
        session.put("user", verifiedUser.id);
        index();
    } else {
        if(!OpenID.id(user).verify()) { // will redirect the user
            flash.error("Cannot verify your OpenID");
            login();
        }
    }
}
```

And the **login.html** template:

```
#{if flash.error}
<h1>${flash.error}</h1>
#{/if}

<form action="@{Application.authenticate()}" method="POST">
    <label for="user">What's your OpenID?</label>
    <input type="text" name="user" id="user" />
    <input type="submit" value="login..." />
</form>
</code>
```

And finally the routes definitions:

```
GET   /                     Application.index
GET   /login                Application.login
*     /authenticate         Application.authenticate
```

The Play framework allows you to easily perform Ajax requests, and is shipped with jQuery by default.
This section describes how to effectively use jQuery within the framework.

The Play framework also comes with the handy **jsAction** tag to transparently get a method definition from the controller.

Using jQuery with the jsAction tag

The #{jsAction /} tag returns a JavaScript function which constructs a URL based on a server action and free variables. It does not perform an AJAX request; these have to be done by hand using the returned URL.

Let's see an example:

```
GET     /hotels/list        Hotels.list
```

Now you can import this route client side:

```
<script type="text/javascript">
   var listAction = #{jsAction @list(':search', ':size', ':page') /}
   $('#result').load(listAction({search: 'x', size: '10', page: '1'}), function() {
       $('#content').css('visibility', 'visible')
   })
</script>
```

In this example we are requesting the list method from the default Application controller. We are also passing three parameters: search, size and page. The request we perform is then saved into the listAction variable. Now using jQuery and the **load** function we are performing a request (an HTTP GET request in fact).

In fact, the following request is sent:

```
GET /hotels/list?search=x&size=10&page=1
```

In that case the request returns HTML data.

However, it is also possible to return JSON or XML and to have jQuery interpret the data. In your controller, use the appropriate render method (renderJSON, renderXML or an XML template).

Please refer to the jQuery documentation for more info.

Also please note that we could perform a POST; the jQuery method should then be changed to:

```
$.post(listAction, function(data) {
  $('#result').html(data);
});
```

The CRUD (Create, Read, Update, Delete) module generates a fully usable web interface for your JPA Model objects.

Let's see a simple example.

Enable the CRUD module for the application

In the **/conf/application.conf** file, enable the CRUD module by adding this line:

```
# The crud module
module.crud=${play.path}/modules/crud
```

Import default CRUD routes

In the **conf/routes** file, import the default module routes by adding this line:

```
# Import CRUD routes
*      /admin              module:crud
```

Note that using the default routes file is not required. You can also define your own routes, or mix the two.

Create a User class

We will start by creating a User class:

```
package models;

import play.*;
import play.db.jpa.*;

import javax.persistence.*;
import java.util.*;

@Entity
public class User extends Model {

        public String name;
        public String email;
        public String address;

}
```

Create the Users controller

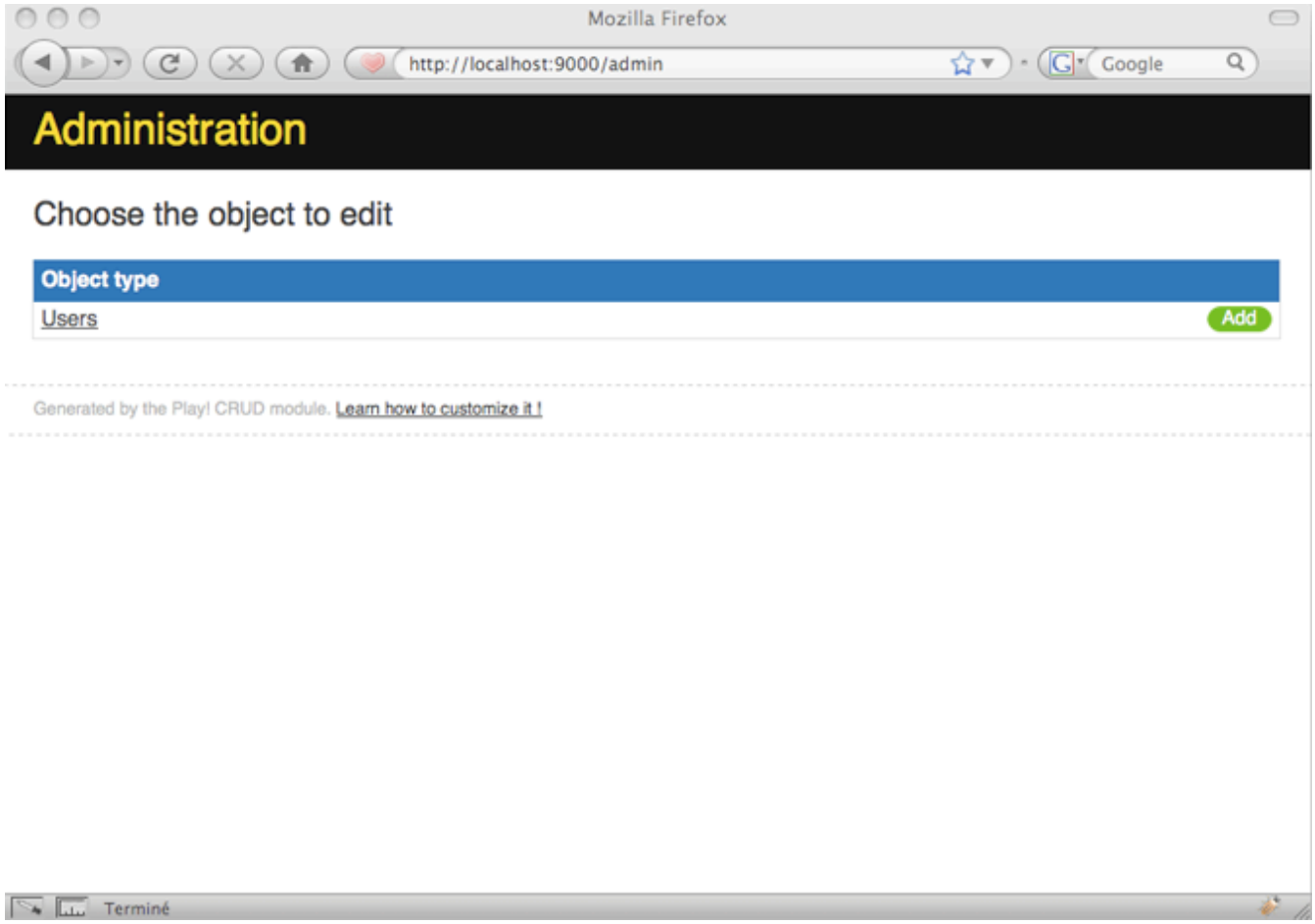
Then, we create a simple controller that just inherits the CRUD controller.

```
package controllers;

public class Users extends CRUD {

}
```

Now open http://localhost:9000/admin and you should see the User admin area.



The controller's class name has to be the model class name with a final 's'. If you want to name it differently, you can do it using an annotation.

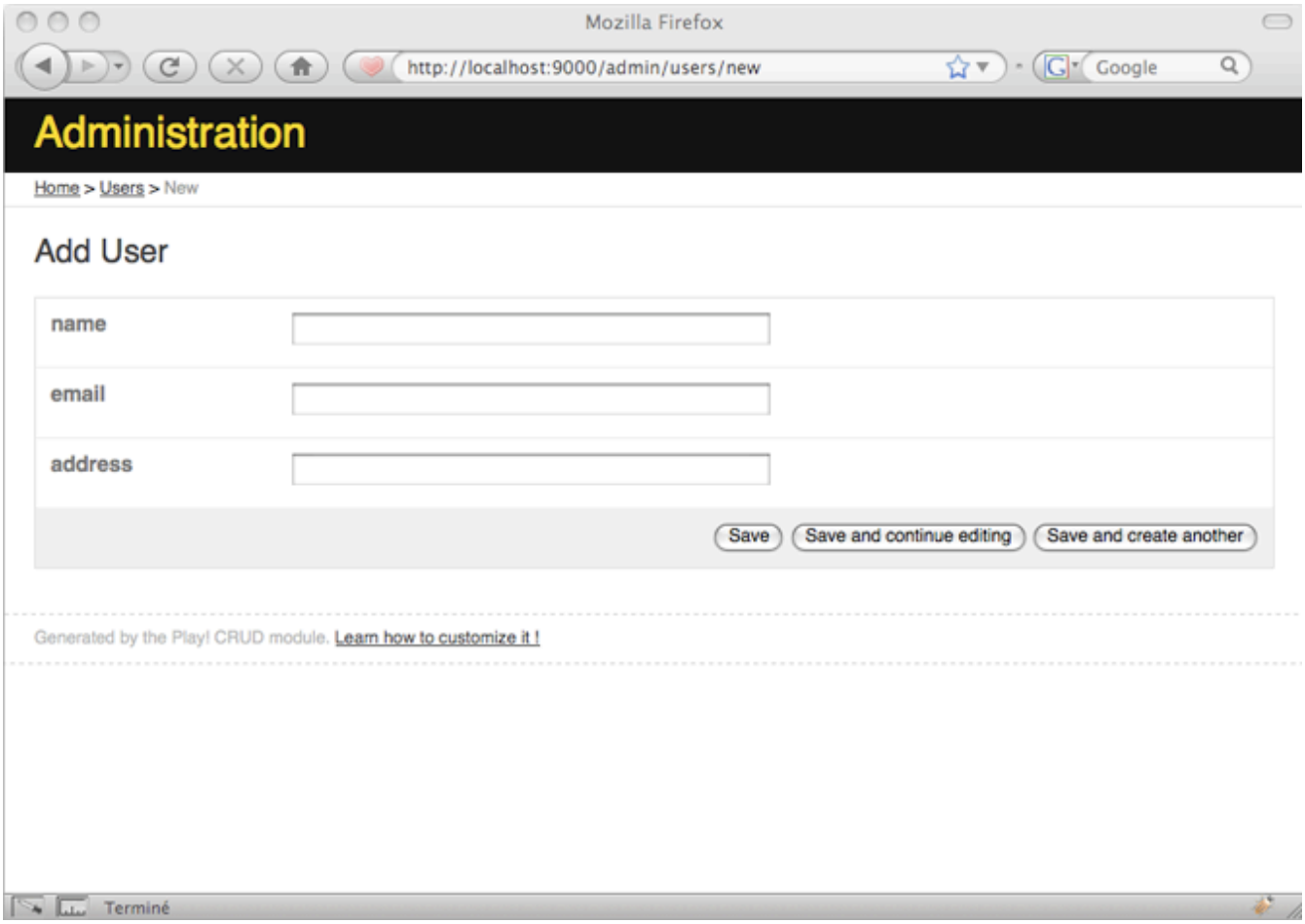
```
package controllers;

@CRUD.For(User.class)
public class AdminUsers extends CRUD {

}
```

The User form

Click the **Add** button, and you should see the User form.



Now we can add some validation rules to the User class:

```
package models;

import play.*;
import play.db.jpa.*;

import javax.persistence.*;
import java.util.*;

import play.data.validation.*;

@Entity
public class User extends Model {

    @Required
    @MinSize(8)
    public String name;

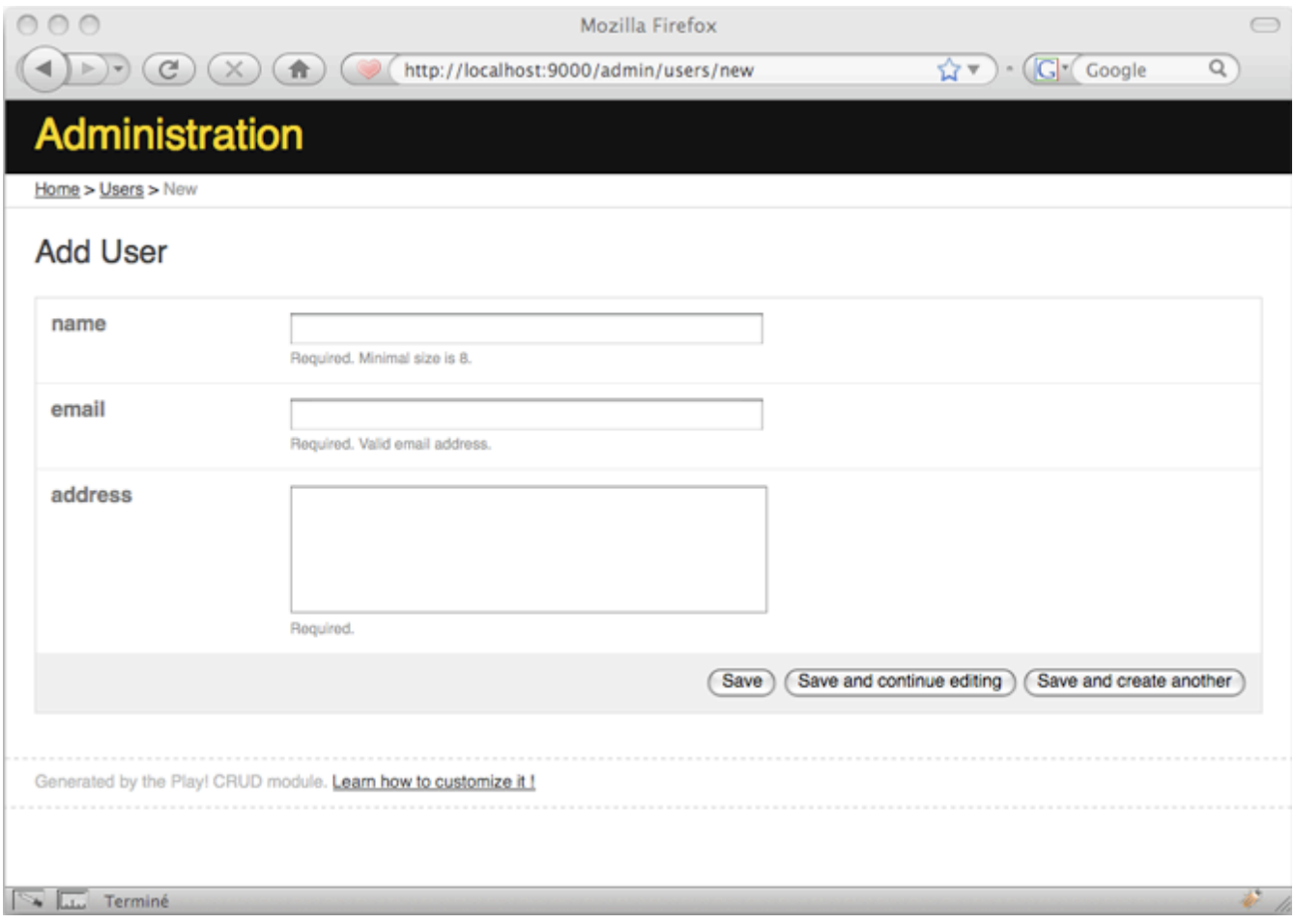
    @Required
    @Email
    public String email;

    @Required
    @MaxSize(1000)
    public String address;

    public String toString() {
        return email;
    }

}
```

Refresh the User form and you will see that the validation is automatically applied.

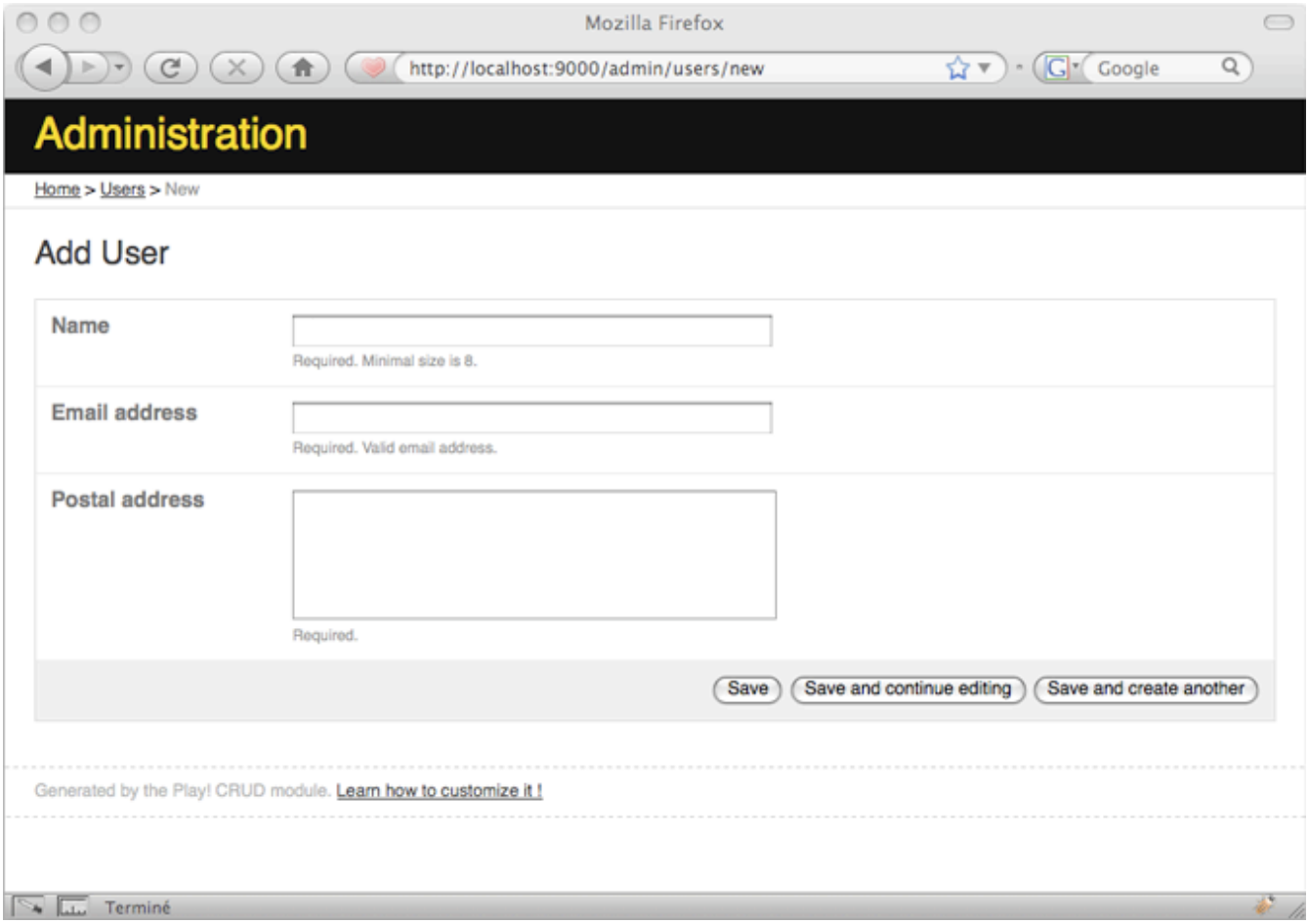


Change the form label

Add these lines to the **conf/messages** file in your application:

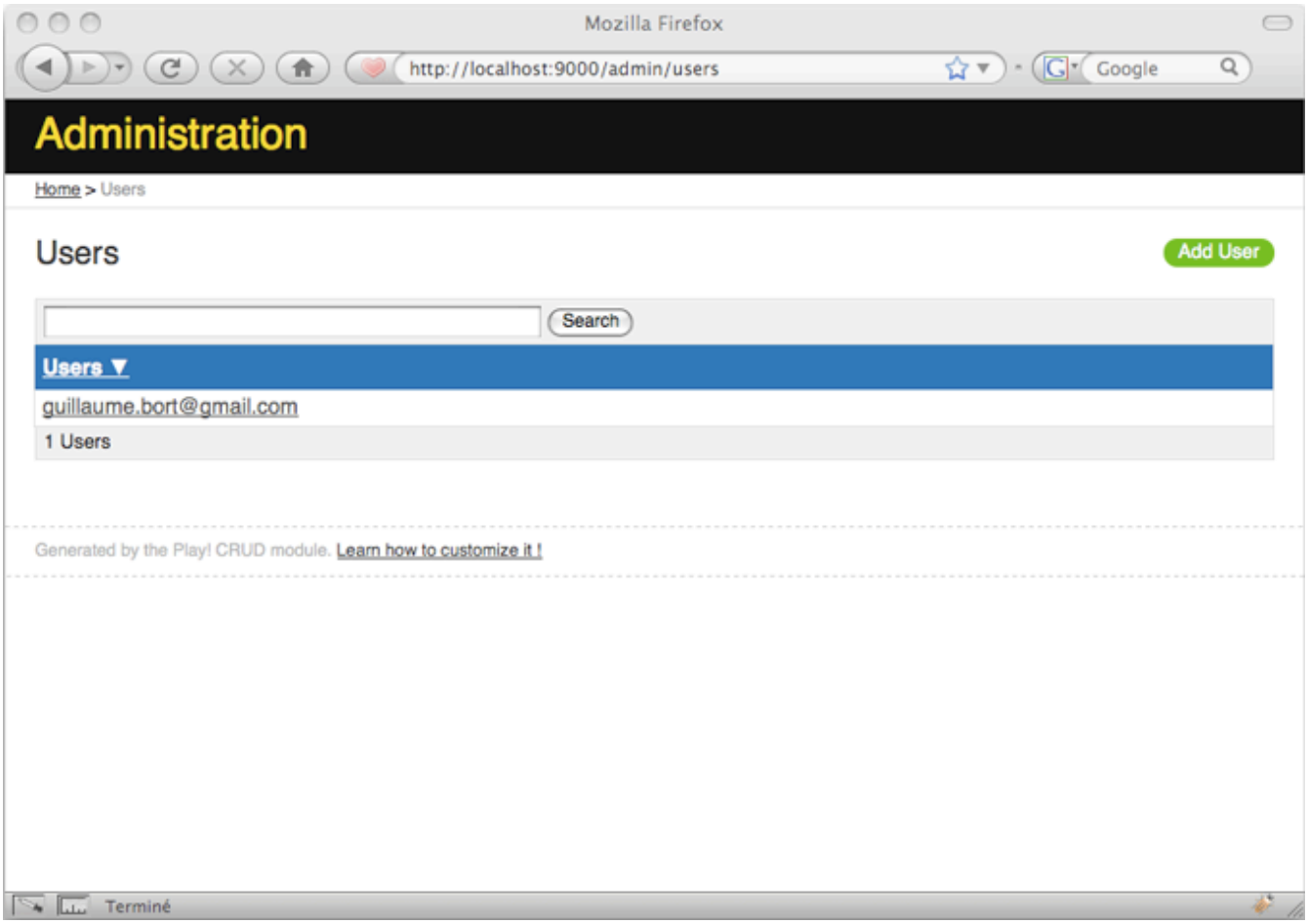
```
name=Name
email=Email address
address=Postal address
```

And refresh the User form.



Create a User and customize the list view

The default list view uses only one column containing the result of the object's toString() method.



To customize this view, we need to create the **/app/views/Users/list.html** template in the application.

Open a shell, go the application directory and type:

```
play crud:ov --template Users/list
```

This will copy the default CRUD **list.html** template to the **Users/list.html** template in your application, overwriting it if present.

Edit the template like this:

```
#{extends 'CRUD/layout.html' /}

<div id="crudList" class="${type.name}">

        <h2 id="crudListTitle">&{'crud.list.title', type.name}</h2>

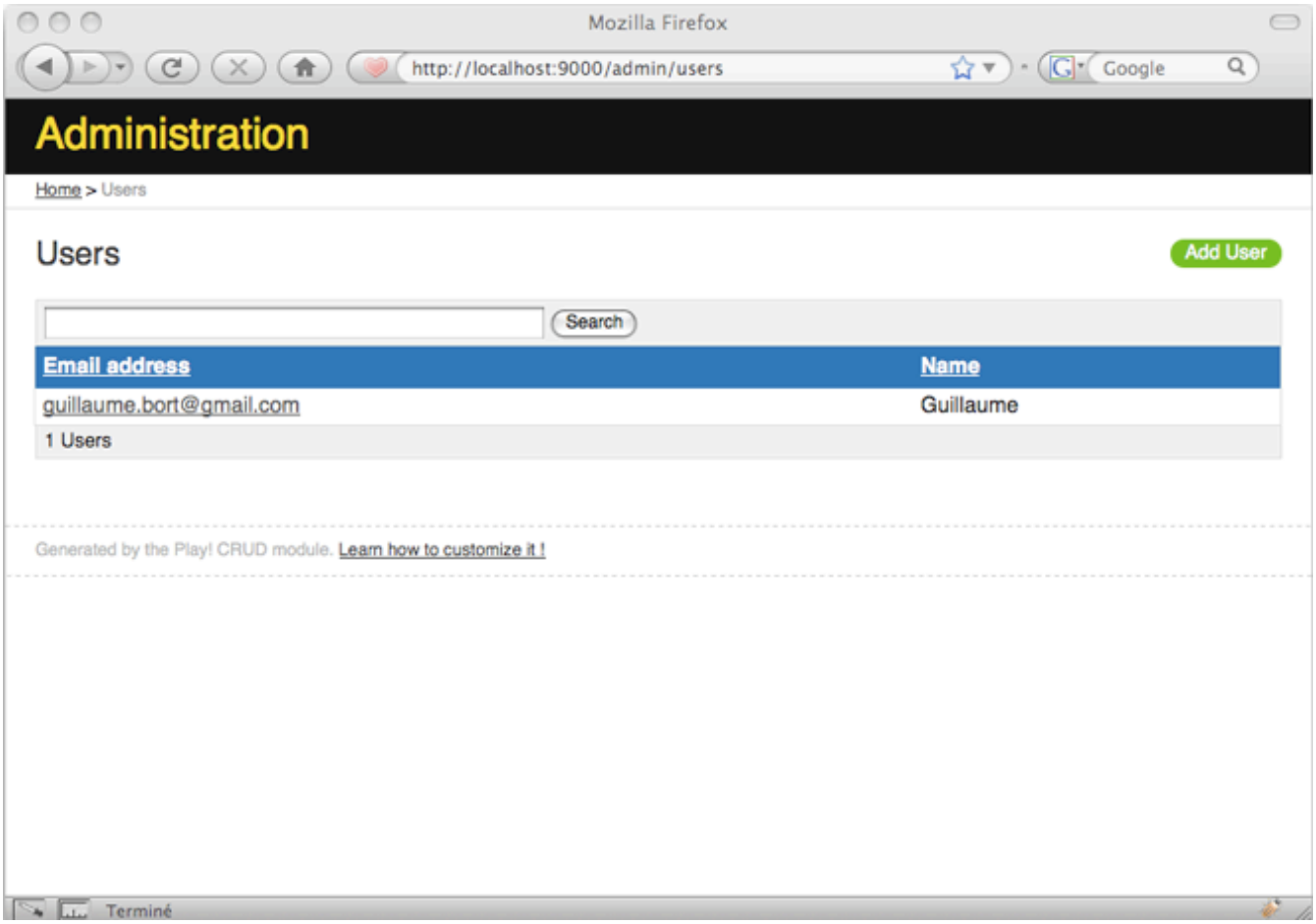
        <div id="crudListSearch">
                #{crud.search /}
        </div>

        <div id="crudListTable">
                #{crud.table fields:['email', 'name'] /}
        </div>

        <div id="crudListPagination">
                #{crud.pagination /}
        </div>

        <p id="crudListAdd">
                <a href="@{blank()}">&{'crud.add', type.modelName}</a>
        </p>
</div>
```

and refresh the list.



Custom fields rendering: the crud.custom tag

You can go a bit further by customizing the way each field of your "User" entity is displayed in the list and form views.

To customize a field, use the "#{crud.custom}" tag:

```
#{crud.table fields:['name', 'company']}

   #{crud.custom 'company'}
       <a
href="@{Companies.show(object.company.id)}">${object.company.name}</a>
   #{/crud.custom}

#{/crud.table}
```

You can also display additional columns or form inputs by defining custom handlers:

```
#{crud.table fields:['name', 'company', 'edit']}

   #{crud.custom 'company'}
       <a
href="@{Companies.show(object.company.id)}">${object.company.name}</a>
   #{/crud.custom}

   #{crud.custom 'edit'}
       <a href="@{Users.edit(object.id)}">Edit</a>
   #{/crud.custom}

#{/crud.table}
```

List of String and List of enumeration

The CRUD module is showing them as a text field. In this text field, the list is represented by a comma separated list of String.
For example:

```
@Entity
public class Account extends Model {

        @CollectionOfElements
        public Set<ContentType> contentTypes;

        @CollectionOfElements
        public Set<String> usernames;

        public Account(Set<String> usernames {
                super();
                this.usernames = usernames;
        }
}
```

This is shown as:

"myEnumId1","myEnumId2" for the contentTypes and "string1","string2" for the usernames. Per definition, this is what you should first customize in your CRUD module.

Limitation

The CRUD module will show the relationships that are unidirectional in one entity: the one that does not have the **mappedBy** attribute.

The simple **Secure** module helps you to set-up basic authentication and authorization management in your application. It provides a simple **controllers.Secure** controller that defines a set of interceptors that you can easily add to your own controllers using the **@With** annotation.

Enable the Secure module for the application

In the **/conf/application.conf** file, enable the Secure module by adding this line:

```
# The secure module
module.secure=${play.path}/modules/secure
```

Import default Secure routes

In the **conf/routes** file, import the default module routes by adding this line:

```
# Import Secure routes
*      /               module:secure
```

Note that it's not required to use the default routes file. You can also define your own routes, or mix the two.

Protecting a controller

To protect a controller you just have to annotate it using **@With**. For example:

```
@With(Secure.class)
public class Application extends Controller {

    public static void index() {
        render();
    }

}
```

This controller will be automatically protected with the default authentication page.

Customize the authentication mechanism

By default, the login page will accept any login/password. To customize it your application has to provide a **Security** provider. Just create a class in the **controllers** package that extends the **controllers.Secure.Security** class. Then you can override the **authenticate(String username, String password)** method.

```
package controllers;

public class Security extends Secure.Security {

    static boolean authenticate(String username, String password) {
        User user = User.find("byEmail", username).first();
        return user != null && user.password.equals(password);
    }

}
```

Note that you can also override other methods to customize how the application should react to authentication events (onAuthenticated, onDisconnected).

Retrieving the connected user

From your application code, you can reuse the Security helper that you've just created to retrieve the connected user.

```
@With(Secure.class)
public class Application extends Controller {

    public static void index() {
        String user = Security.connected();
        render(user);
    }

}
```

Adding authorization check

You can use the **@Check** annotation either on controller classes or action methods to tell the Secure module to check that the connected user has the required authorization to call this action.

For example:

```
@With(Secure.class)
public class Application extends Controller {

   ...

   @Check("isAdmin")
   public static void delete(Long id) {
       ...
   }

}
```

By default the secure module will always authorize all checks. You have to customize by overriding one more method in your **Security** class.

```
package controllers;

public class Security extends Secure.Security {

    ...

    static boolean check(String profile) {
        User user = User.find("byEmail", connected()).first();
        if ("isAdmin".equals(profile)) {
            return user.admin;
        }
        else {
            return false;
        }
    }

}
```

These are the standard tags built in the template engine.

> There is a separate Java extensions manual page.

a

The **a** tag inserts an HTML link to a controller action.

```
#{a @Application.logout()}Disconnect#{/a}
```

Rendered as:

```
<a href="/application/logout">Disconnect</a>
```

If the action you try to call does not have any route able to invoke it using a GET method, Play will automatically generate a hidden form that will be submitted on link click using JavaScript.

doLayout

Used with template inheritance, this tag inserts the evaluated sub-template's contents.

```
<!-- common header here -->
<div id="content">
    #{doLayout /}
</div>
<!-- common footer here -->
```

else

Of course used with the **if** tag.

```
#{if user}
    Connected user is ${user}
#{/if}
#{else}
    Please log in
#{/else}
```

However, you can also use it with the **list** tag and it will be executed if the list is empty.

```
#{list items:task, as:'task'}
    <li>${task}</li>
#{/li}

#{else}
    Nothing to do...
#{/else}
```

elseif

```
#{if tasks.size() > 1}
    Busy tasklist
#{/if}

#{elseif tasks}
    One task on the list
#{/elseif}

#{else}
    Nothing to do
#{/else}
```

As for **else**, you can use it with the **list** tag.

errors

Iterates over the current validation errors.

```
<ul>
#{errors}
    <li>${error}</li>
#{/errors}
</ul>
```

The tag defines implicit variables in its body.

- error_index, the error's index, starting at 1
- error_isLast, true for the last element
- error_isFirst, true for the first element
- error_parity, alternates between odd and even

```
<table>
<tr><th>#</th><th>Error</th></tr>
#{errors}
    <tr
class="${error_parity}"><td>${error_index}</td><td>${error}</td></tr> #{/errors} </table
```

extends

Makes the template inherit another template.

```
#{extends 'main.html' /}
```

field

The field tag is a helper, based on the spirit of the Don't Repeat Yourself. It works this way:

Instead of writing this:

```
<p>
  <label>&{'user.name'}</label>
  <input type="text" id="user_name" name="user.name" value="${user?.name}"
class="${errors.forKey('user.name') ? 'has_error' : ''}">
  <span class="error">${errors.forKey('user.name')}</span>
</p>
```

You can just write:

```
#{field 'user.name'}
<p>
  <label>&{field.name}</label>
  <input type="text" id="${field.id}" name="${field.name}" value="${field.value}"
class="${field.errorClass}">
  <span class="error">${field.error}</span>
</p>
#{/}
```

So you don't repeat **user.name** again and again.

form

Inserts a **form** tag. The HTTP method is guessed from the route, and defaults to POST. If there are both GET and POST routes configured for the URL, the tag will default to using the first route defined in routes.

- Optional id attribute sets an ID to the form element.
- Optional enctype attribute sets the form's data encoding. it defaults to 'application/x-www-form-urlencoded'.

Charset encoding is always 'utf-8'.

```
#{form @Client.create() , id:'creationForm' enctype:'multipart/form-data' }
   ...
#{/form}
```

Rendered as:

```
<form action="/client/create" id="frm" method="POST" accept-charset="utf-8"
enctype="multipart/form-data">
 ...
</form>
```

get

Retrieves a value defined with a **set** tag. You may use the get/set mechanism to exchange values between templates, layouts and sub-templates.

```
<head>
    <title>#{get 'title' /}
</head>
```

You can also use the get tag in the following way, which will display "Homepage" if title has not been specified.

```
<head>
    <title>#{get 'title'}Homepage #{/}
</head>
```

i18n

Exports localized messages in JavaScript. Localized messages are available from your JavaScript code using the **i18n()** function.

Define your translations in the **conf/messages** file.

```
hello_world=Hello, World !
hello_someone=Hello %s !
```

Include the messages in your template (or layout) page:

```
#{i18n /}
```

And retrieve keys from JavaScript:

```
alert(i18n('hello_world'));
alert(i18n('hello_someone', 'John'));
```

if

Evaluates the given test, and if true, evaluates the tag body.

```
#{if user.countryCode == 'en"' }
    Connected user is ${user}
#{/if}
```

Using composite conditions:

```
#{if ( request.actionMethod == 'administer'  && user.isAdmin() ) }
    You are admin, allowed to administer.
#{/if}
```

ifnot

Cleaner alternative to #{if !condition}

```
#{ifnot tasks}
    No tasks today
#{/ifnot}
```

include

Includes another template. All of the current template's variables are directly available in the included template.

```
<div id="tree">
    #{include 'tree.html' /}
</div>
```

jsAction

The #{jsAction /} tag returns a JavaScript function which constructs a URL based on a server action and free variables. It does not perform an AJAX request; these have to be done by hand using the returned URL.

Let's see an example:

```
GET     /users/{id}         Users.show
```

Now you can import this route client side:

```
<script type="text/javascript">
    var showUserAction = #{jsAction @Users.show(':id') /}

    var displayUserDetail = function(userId) {
        $('userDetail').load( showUserAction({id: userId}) )
    }
</script>
```

list

Iterates over an object collection.

```
<ul>
#{list items:products, as:'product'}
    <li>${product}</li>
#{/list}
</ul>
```

The tag defines implicit variables in its body. The variable names are prefixed with the loop variable name.

- name_index, the item's index, starting at 1
- name_isLast, true for the last element
- name_isFirst, true for the first element
- name_parity, alternates between odd and even

```
<ul>
#{list items:products, as:'product'}
    <span class="${product_parity}">${product_index}. ${product}</span>
    ${product_isLast ? '' : '-'}
#{/list}
</ul>
```

The **items** parameter is optional and can be replaced by the default **arg** argument.

So you can rewrite:

```
#{list items:users, as:'user'}
    <li>${user}</li>
#{/list}
```

as:

```
#{list users, as:'user'}
    <li>${user}</li>
#{/list}
```

for loops are easy to create using Groovy **range** object:

```
#{list items:0..10, as:'i'}
    ${i}
#{/list}
```

```
#{list items:'a'..'z', as:'letter'}
    ${letter} ${letter_isLast ? '' : '|' }
#{/list}
```

The **as** parameter is optional as well. It uses _ as default variable name:

```
#{list users}
    <li>${_}</li>
#{/list}
```

script

Inserts a **script** tag in the template. By convention, the tag refers to a script in **/public/javascripts**

- src (required) - script file name, without the leading path /public/javascripts
- id (optional) - an id attribute value for the generated script tag
- charset (optional) - sets source encoding - defaults to UTF-8

The **src** parameter can be replaced by the default **arg** argument.

```
#{script 'jquery-1.4.2.min.js' /}
#{script id:'datepicker' , src:'ui/ui.datepicker.js', charset:'utf-8' /}
```

set

Define a value which can be retrieved in the same template or any layout with the **get** tag.

```
#{set title:'Admin' /}
#{set style:'2columns' /}
```

You can also use variables:

```
#{set title:'Profile of ' + user.login /}
```

You can define the value of variables in the body:

```
#{set 'title'}
    Profile of ${user.login}
#{/set}
```

stylesheet

Inserts a **link** tag in the template. By convention, the tag refers to a CSS file in **/public/stylesheets**

- src (required) - file name, without the leading path /public/stylesheets
- id (optional) - an id attribute value for the generated link tag
- media (optional) - a media attribute value: screen, print, aural, projection…
- title (optional) - title attribute value (or description)

The **src** parameter can be replaced by the default **arg** argument.

```
#{stylesheet 'default.css' /}
#{stylesheet id:'main', media:'print', src:'print.css', title:'Print stylesheet' /}
```

verbatim

Disables HTML escaping in template output, like the raw() Java extension, but for the whole tag body.

```
${'&amp;'}
#{verbatim}${'&amp;'}#{/verbatim}
```

In this example, the first line outputs **&** while the second line outputs an ampersand.

Java extensions add convenience methods to objects for use in view templates, making expressions more expressive.

The code examples show an example template expression, with its result on the next line.

Collection extensions

join(separator)

Concatenates the collection's entries, with the given separator between each entry. Returns: **String**.

```
${['red', 'green', 'blue'].join('/')}
red/green/blue
```

pluralize()

Returns an 's' when the collection's size is not 1. Returns: **String**.

```
colour${['red', 'green', 'blue'].pluralize()}
colours
```

pluralize(plural)

Returns the given plural when the collection's size is not 1. Returns: **String**.

```
box${['red', 'green', 'blue'].pluralize('es')}
boxes
```

pluralize(singular, plural)

Returns the given plural form when the collection's size is not 1; returns the given singular form when it is 1. Returns: **String**.

```
journ${['red'].pluralize('al', 'aux')}
journal

journ${['red', 'green', 'blue'].pluralize('al', 'aux')}
journaux
```

Date extensions

format(format)

Formats the date using the given date format pattern. Returns: **String**.

```
${new Date(1275910970000).format('dd MMMM yyyy hh:mm:ss')}
07 June 2010 01:42:50
```

format(format, language)

Formats the date using the given date format pattern, in the given language. Returns: **String**.

```
${new Date(1275910970000).format('dd MMMM yyyy hh:mm:ss', 'fr')}
07 juin 2010 01:42:50
```

since()

Formats the date as a relative time, compared to now, e.g. **3 minutes ago**. Returns: **String**.

```
${new Date(new Date().getTime() - 1000000).since()}
16 minutes ago
```

You can customise the output using the following messages: **since.seconds**, **since.minutes**, **since.hours**, **since.days**, **since.months** and **since.years**.

since(condition)

Formats the date as a relative time, compared to now. If the condition is true, dates more than one month ago are formatted as a date. Returns: **String**.

You can customise the output using the same messages as for **since()**, as well as **since.format** for dates more than one month ago.

```
${new Date(1262350000000).since(false)}
5 months ago

${new Date(1262350000000).since(true)}
Jan 1, 2010
```

Long extensions

asdate(format)

Formats a time stamp as a date. Returns: **String** - a formatted date.

```
${1275910970000.asdate('dd MMMM yyyy hh:mm:ss')}
07 June 2010 01:42:50
```

asdate(format, language)

Formats a time stamp as a date, in the given language. Returns: **String** - a formatted date.

```
${1275910970000.asdate('dd MMMM yyyy hh:mm:ss', 'fr')}
07 juin 2010 01:42:50
```

formatSize()

Formats a number of bytes as a file size, with units.

```
${726016L.formatSize()}
709KB
```

Map extensions

asAttr()

Formats the map's keys and values as HTML attributes. Returns: **play.templates.Template.ExecutableTemplate.RawData**.

```
${[id:'42', color:'red'].asAttr()}
id="42" color="red"
```

asAttr(condition)

Formats the map's keys and values as HTML attributes, if the condition is true. Returns: **play.templates.Template.ExecutableTemplate.RawData**.

```
${[id:'42', color:'red'].asAttr(true)}
id="42" color="red"
```

Number extensions

divisibleBy(divisor)

Returns true if the number is divisible by the given number - the divisor. Returns: **boolean**.

```
${42.divisibleBy(7)}
true
```

format(format)

Formats the number using the given number format pattern. Returns: **String**.

```
${42.format('000.00')}
042.00
```

formatCurrency(currencyCode)

Formats the number as a currency for the given currency code, e.g. **EUR**. Returns: **String**.

```
${42.formatCurrency('EUR').raw()}
&euro; 42.00
```

page(pageSize)

Returns the page number, for the given page size, from interpreting the number as an index. Returns: **Integer**.

```
${42.page(10)}
5
```

pluralize()

Returns an 's' when the number is not 1. Returns: **String**.

```
colour${['red', 'green', 'blue'].pluralize()} - colour${3.pluralize()}
colours - colours
```

pluralize(plural)

Returns the given plural when the number is not 1. Returns: **String**.

```
box${3.pluralize('es')}
boxes
```

pluralize(singular, plural)

Returns the given plural form when the number is not 1; returns the given singular form when it is 1. Returns: **String**.

```
journ${1.pluralize('al', 'aux')}
journal

journ${3.pluralize('al', 'aux')}
journaux
```

Object extensions

addSlashes()

Backslash-escapes Java-escaped single and double quotes in the object's **String** representation. Returns: **String**.

```
${"single quote (')".addSlashes().raw()} ${'double quote (")'.addSlashes().raw()}
single quote (\') double quote (\")
```

capAll()

Capitalises every word in the object's **String** representation. Returns: **String**.

```
${"lorum ipsum dolor".capAll()}
Lorum Ipsum Dolor
```

capFirst()

Capitalises the first word in the object's **String** representation. Returns: **String**.

```
${"lorum ipsum dolor".capFirst()}
Lorum ipsum dolor
```

cut(substring)

Removes occurrences of the given sub-string. Returns: **String**.

```
${"lorum ipsum dolor".cut('um')}
lor ips dolor
```

escape()

Escapes reserved HTML characters in the object's **String** representation. Returns: **String**.

```
${"The <blink>tag</blink> is evil".escape().raw()}
The <blink>tag</blink> is evil
```

nl2br()

Replaces new-line characters with HTML **br** tags. Returns: **String**.

```
${"one\ntwo".nl2br()}
one<br/>two
```

raw()

Returns the object without template escaping. Returns: **play.templates.Template.ExecutableTemplate.RawData**.

```
${'<'}
<

${'<'.raw()}
<
```

raw(condition)

Returns the object without template escaping, if the condition is true. Returns: **play.templates.Template.ExecutableTemplate.RawData**.

```
${'<'.raw(true)}
<
```

yesNo('yes', 'no')

Returns the first parameter ('yes') if the object evaluates to true, or the second parameter ('no') otherwise. Returns: **String**.

```
${"".yesno('yes', 'no')}
no

${"not empty".yesno('yes', 'no')}
yes
```

String extensions

asXml()

Parses the given XML string. Returns: **groovy.util.slurpersupport.GPathResult**.

camelCase()

Formats the string in camel case, as if for a Java class name. Returns: **String**.

```
${"lorum ipsum dolor".camelCase()}
LorumIpsumDolor
```

capitalizeWords()

Capitalises every word in the string. Returns: **String**.

```
${"lorum ipsum dolor".capitalizeWords()}
Lorum Ipsum Dolor
```

escapeHtml()

Escapes reserved HTML characters. Returns: **String**.

```
${"The <blink>tag</blink> is evil".escape().raw()}
The <blink>tag</blink> is evil
```

escapeJavaScript()

Escapes reserved JavaScript characters. Returns: **String**.

```
${"single quote (') double quote (\")".escapeJavaScript().raw()}
single quote (\') double quote (\")
```

escapeXml()

Escapes reserved XML characters. Returns: **String**.

```
${"<>\"&".escape().raw()}
<>&quot;&amp;
```

last()

Returns the last item in the list. Returns: **Object**.

```
${['red', 'green', 'blue'].last()}
blue
```

noAccents()

Removes accents from the letters in the string. Returns: **String**.

```
${"Stéphane Épardaud".noAccents()}
Stephane Epardaud
```

pad(length)

Pads the string with ** ** up to the given length. Returns: **String**.

```
${"x".pad(4).raw()}
x   
```

slugify()

Formats the string as a 'slug' for use in URLs, that avoids reserved URL path characters. Returns: **String**.

```
${"The Play! framework's manual".slugify()}
the-play-framework-s-manual
```

urlEncode()

Escapes reserved URL query string characters. Returns: **String**.

```
${"!@#\$%^&()".urlEncode()}
%21%40%23%24%25%5E%26%28%29
```

String array extensions

add(value)

Adds value to the end of the array. Returns: **String[]**.

```
${(["red", "green", "blue"] as String[]).add('pink').join(' ')}
red green blue pink
```

contains(string)

Returns true if the array contains the given string. Returns: **boolean**.

```
${(['red', 'green', 'blue'] as String[]).contains('green')}
true
```

remove(string)

Returns the array, with the given string removed. Returns: **String[]**.

```
${(['red', 'green', 'blue'] as String[]).remove('green').join(' ')}
red blue
```

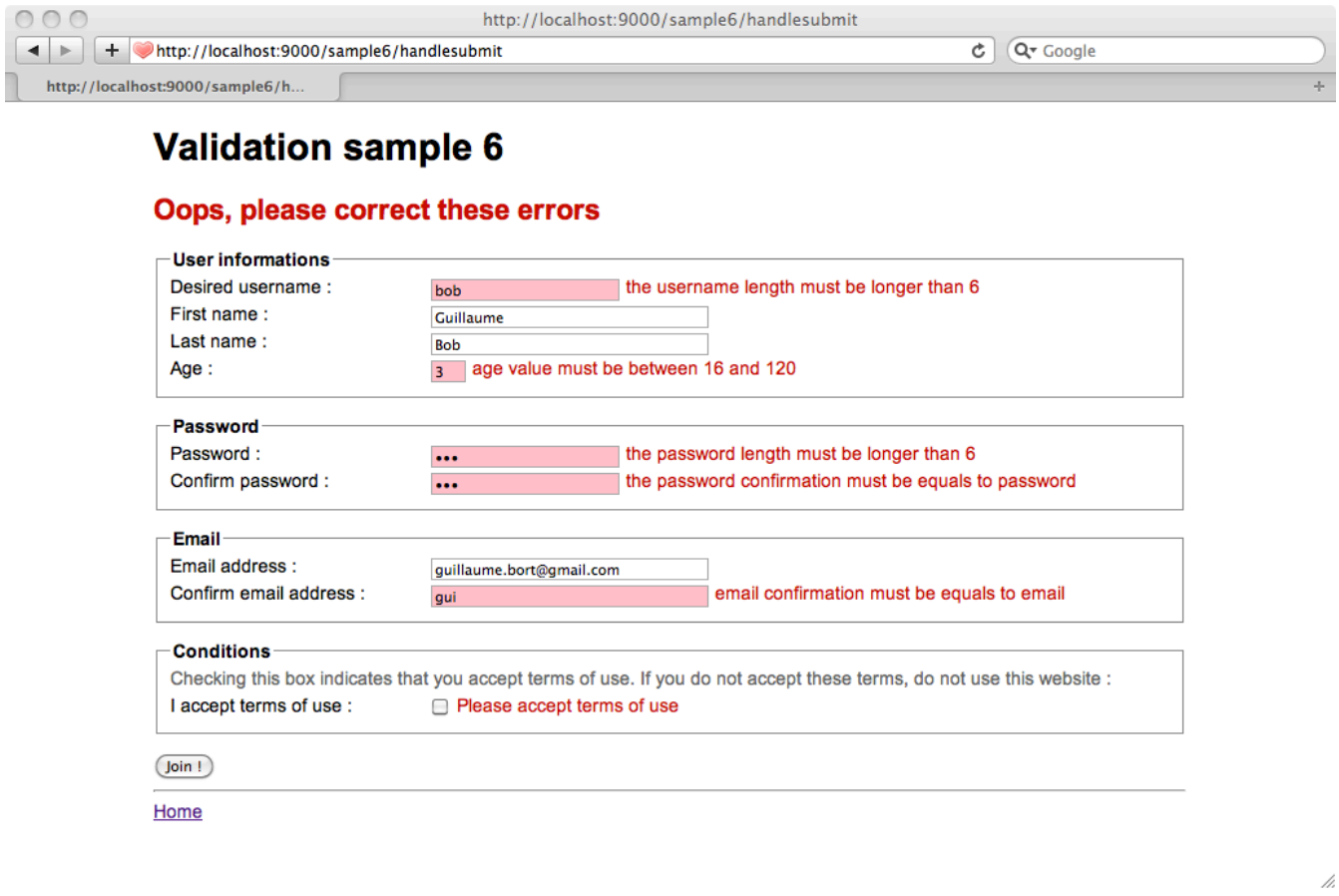
Play 1.0.3 is a maintenance release of the Play 1.0 branch. There is no new major feature in this version, as the development focus is now set to the 1.1 version. The documentation has been updated with missing references, the yabe tutorial now includes a new chapter about **i18n**, and a new sample application has been added.

> **Play 1.0.3** is a maintenance release and is fully compatible with the 1.0 series. If you encounter any problems please ask us on the Google Group.

You can read about the fixed bugs on the 1.0.3 road map page. The most important changes are highlighted on this page.

New sample application: validation

A new sample application has been added. It demonstrates the different ways to create a form with the Play validation support.



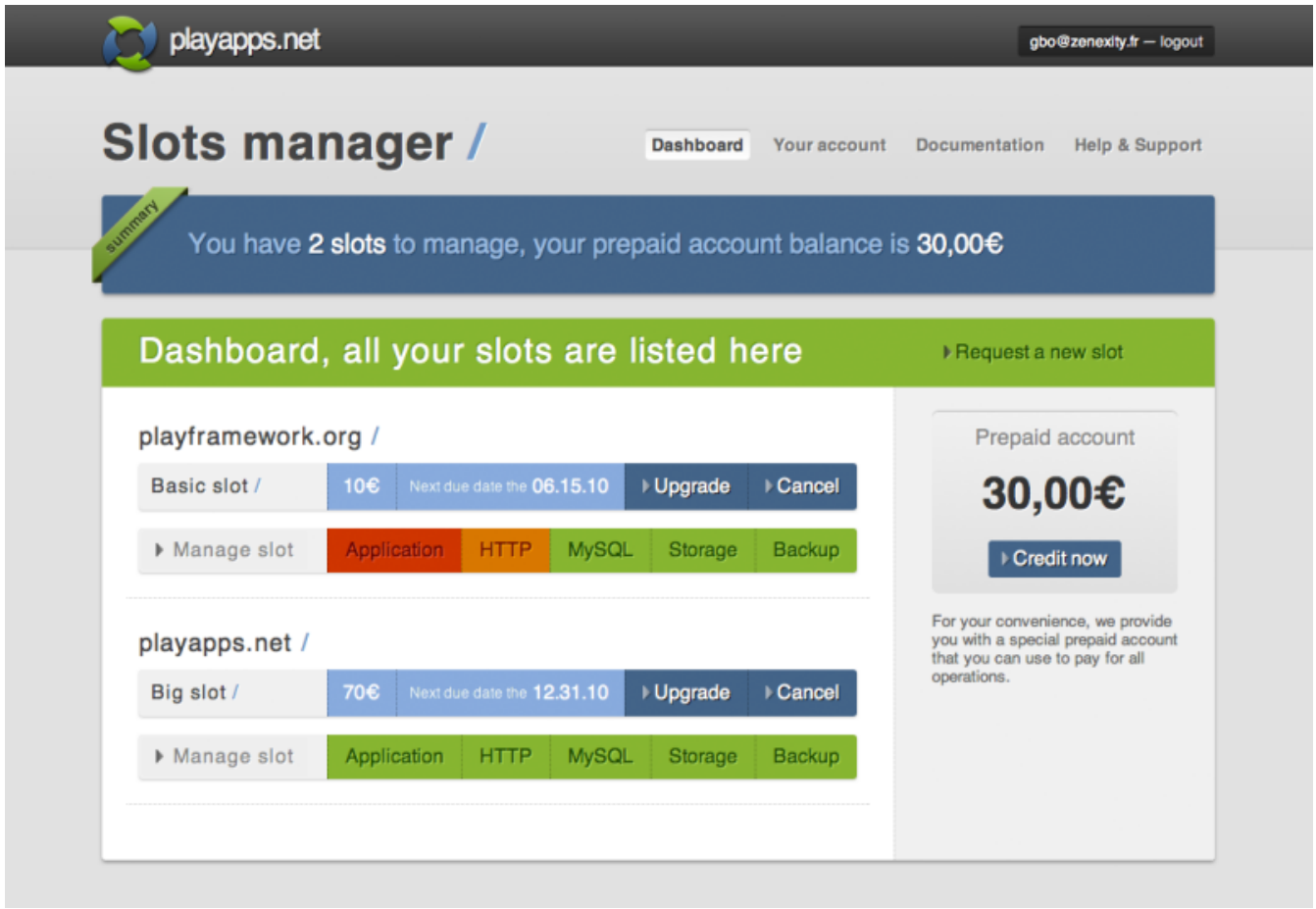
New yabe tutorial chapter: i18n

This complete the existing yabe tutorial with application internationalization and language localisation. Although we could have done this from the start, it is more realistic to build the first version of an application in a single language, and add multiple languages later.

Read the new chapter

playapps.net support

The 1.0.3 release is the first one to be officially supported by the new http://www.playapps.net on the could hosting platform.



Play 1.0.2 is a maintenance release of the Play 1.0 branch. The major new features are the support for the module repository and built-in protection against CSRF attacks. It also fixes a lot of small bugs.

Play 1.0.2 is a maintenance release and is fully compatible with the 1.0 series. If you encounter any problems please ask us on the Google Group.

You can read about the fixed bugs on the 1.0.2 road map page. The most important changes are highlighted on this page.

Module repository

The module repository's goal is to centralize all contributed modules for the Play framework, and allow easy installation. The new module-related commands are:

- play list-modules, to list the repository's contents
- play install, to install a version of a module locally
- play new-module, to create a new module skeleton
- play build-module, to package a module and publish it to the repository.

Also, you will see that almost all modules have been deleted. Only a bunch of 'core' modules are directly available: testrunner, docviewer, crud and secure.

The other modules are now optional. So if you want for example to install the GWT support, just type **play install gwt**, and you will get the latest version of the module.

Why have we moved modules? Because we need to focus on the core framework to have a simple project to manage. Also, as many people wanted to contributed modules, it is easier to do it this way: each module is a standalone project with a dedicated maintainer. So please, if you want to report a bug on a specific module, just use its project homepage and its dedicated bug tracker.

As an immediate benefit, the lifecycle of modules is no more tied to the framework lifecycle. A module can be released more often than the framework itself. And finally, because the framework does not contain optional modules anymore, the Play distribution size has been halved.

Read more about the modules repository on this dedicated page.

Built-in Cross-Site Request Forgery protection

CSRF attacks can be really problematic in web applications.

> This attack method works by including malicious code or a link in a page that accesses a web application that the user is believed to have authenticated. If the session for that web application has not timed out, an attacker may execute unauthorized commands.

To prevent this attack, the first thing to do is to use GET and POST method appropriately. That means that only POST methods must be used to run an interaction that can change the application's state.

For POST requests, the only way to secure critical actions properly is to issue an authenticity token. Play 1.0.2 now has a built-in helper to handle that:

- a new checkAuthenticity() method available in controllers, that checks for a valid authenticity token in the request parameters and sends a forbidden response if something is bad
- session.getAuthenticityToken() generates an authenticity token that is only valid for the current session
- #{authenticityToken /} creates an hidden input field that you can add to any form.

So for example:

```
public static destroyMyAccount() {
    checkAuthenticity();
    ...
}
```

Will only work when called from a form including a proper authenticity token:

```
#{form @ destroyMyAccount()}
    #{authenticityToken /}
    <input type="submit" value="destroy my account">
#{/form}
```

You can of course add this as a before filter if you want to protect all actions of a hierarchy of controllers. Read more in the Security Guide.

HEAD method supported by default

Play now responds automatically to HEAD requests if there is an existing route for the GET method. This is because it is requested by the HTTP RFC that any resource respond to HEAD request as well.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

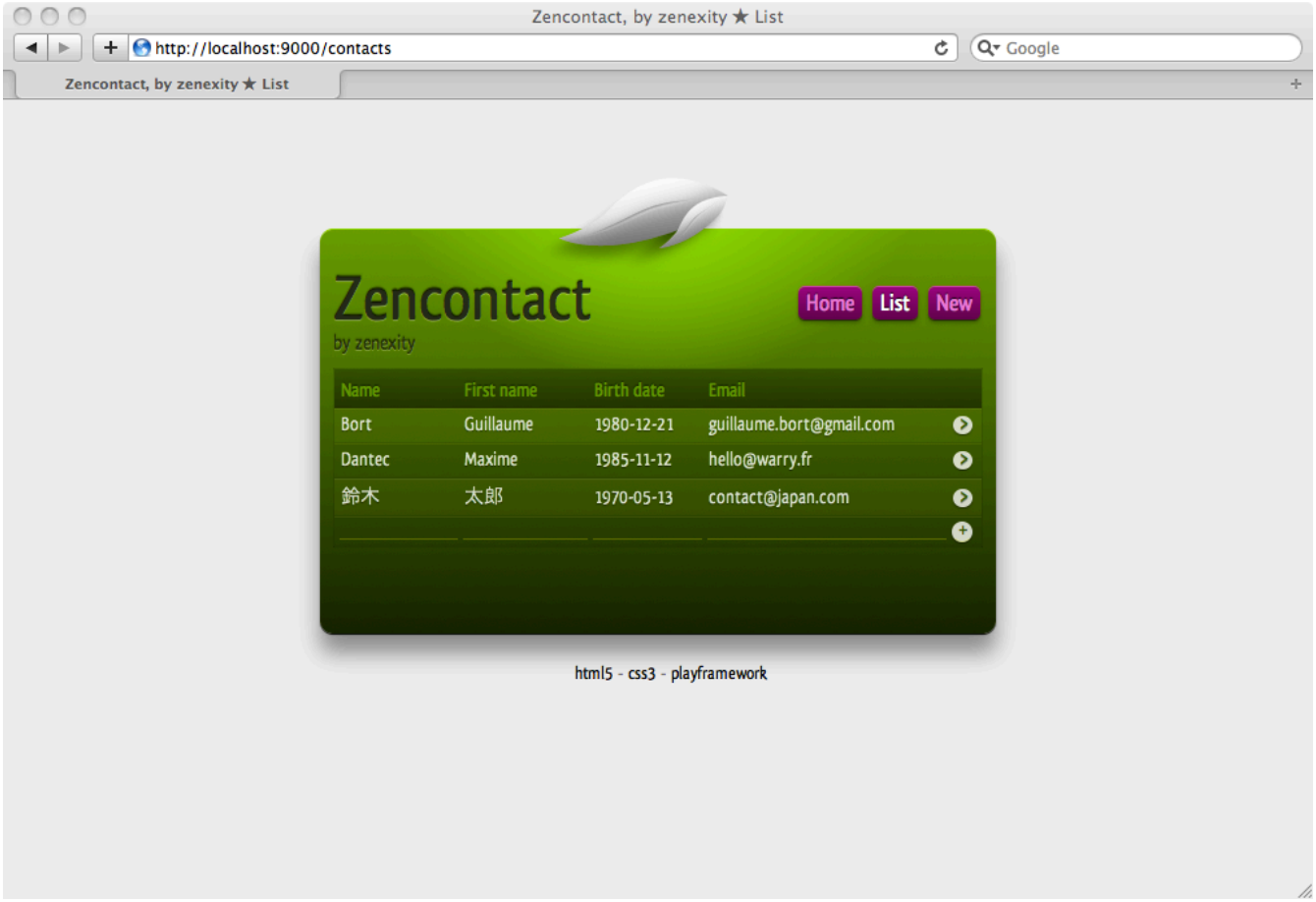
> The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

So any HEAD request will invoke your action method, but the response content will not be sent to the client. You can of course specialize it by adding a custom route to respond to HEAD requests with your routes file. For example:

```
GET     /orders/{id}          Orders.show
HEAD    /orders/{id}          Orders.showHead
```

New sample application, 'zencontact'

A new sample, that is a port of a Wicket-based contact management application. There is also a Scala version of this application available in the Scala module package, for those interested.



Better support for application server deployment

We have tested the deployment of a Play generated WAR archive on several application servers. You can check the current compatibility matrix.

JBoss 4.2.x	JBoss 5.x	JBoss 6M2	Glasshfish v3	IBM Websphere 6.1	IBM Websphere 7	Geronimo 2.x	Tomcat 6.x	Jetty 7.x	Resin 4.0.5

New features for reversed action in templates

The @@ syntax, that allow to reverse an action to an **absolute** URL, is now available in tag parameters. For example:

```
#{form @@save()}
...
#{/}
```

It is very useful if you use the template engine to generate stuff like e-mail, in which you need to express URL in an absolute fashion.

Also, binding complex objects in now supported. So for example, with this action:

```
public static void search(SearchParams params) {
    ...
}
```

SearchParams, being:

```
public class SearchParams {
    public String keywords;
    public String mode;
}
```

You can use in a template, something like:

```
@{search(params)}
```

That would generate a URL containing several values in the queryString, like:

```
/search?params.keywords=xxxx&params.mode=AND
```

A new command to generate the application Javadoc

Now you can generate the **Javadoc** for your project easily, using:

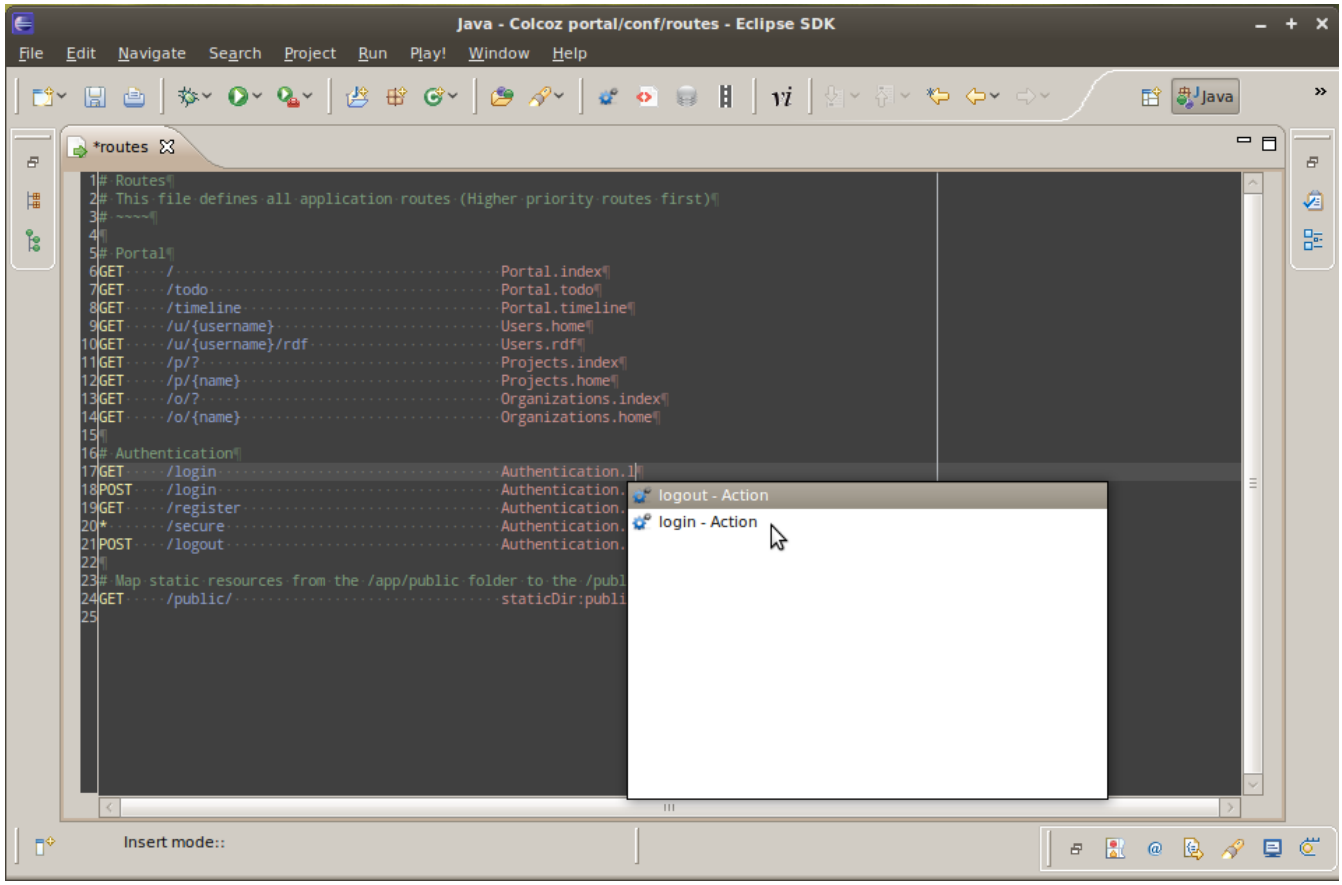
```
play javadoc
```

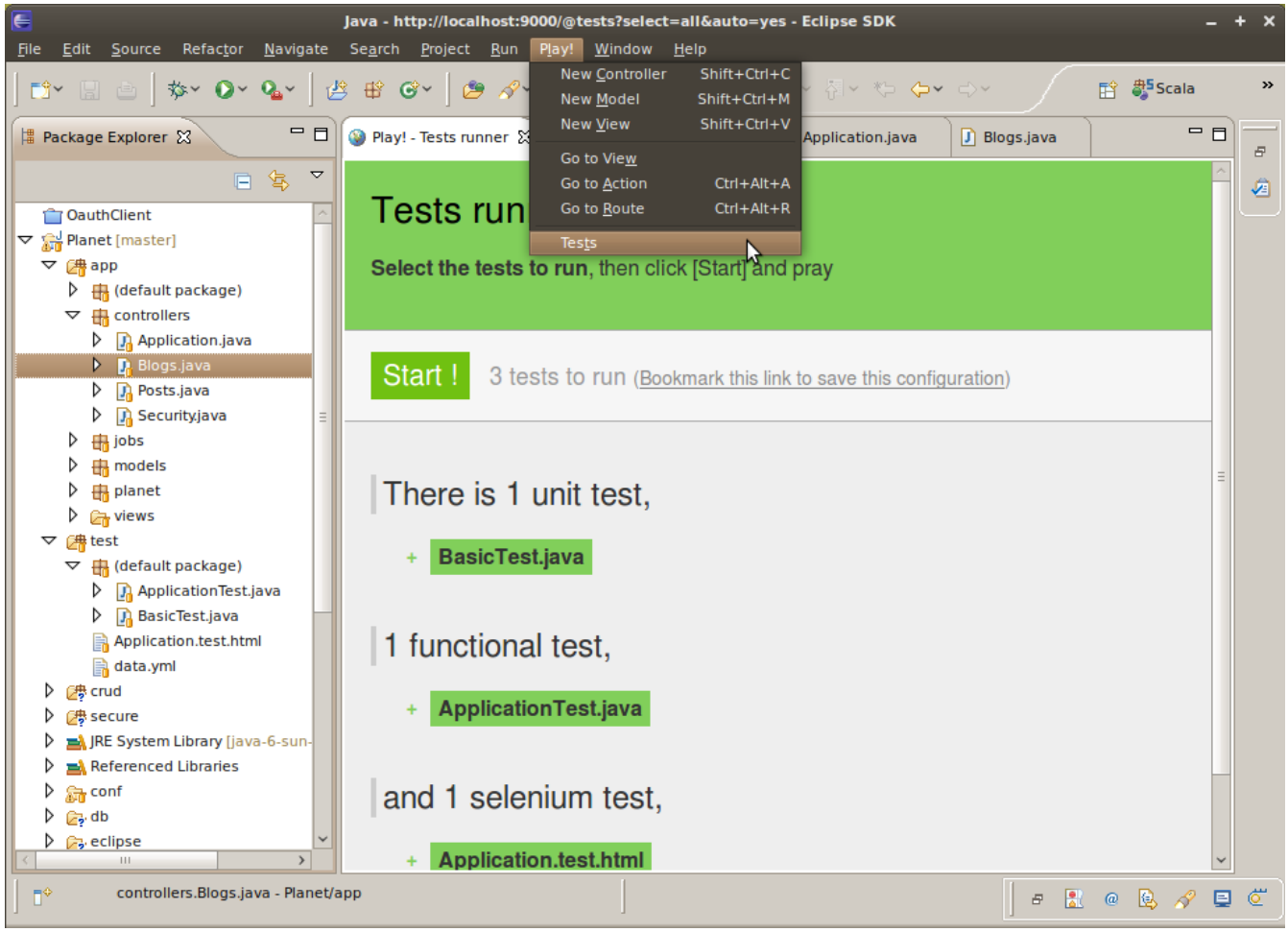
It generates the Javadoc API documentation for your project and its modules.

An even better Eclipse plugin in this release

The Eclipse plugin now has several new features:

- Advanced route file editor, with content assist, missing action detection, and hyperlinks
- Advanced template files editor, with some content assist, and hyperlinks
- Wizard to create new Controllers, Models and Templates
- Integrated test runner (using an embedded web browser)


Java - Colcoz portal/conf/routes - Eclipse SDK
File Edit Navigate Search Project Run Play! Window Help
Java
*routes
Routes
This file defines all application routes (Higher priority routes first)
~~~~
Portal
GET / Portal.index
GET /todo Portal.todo
GET /timeline Portal.timeline
GET /u/{username} Users.home
GET /u/{username}/rdf Users.rdf
GET /p/? Projects.index
GET /p/{name} Projects.home
GET /o/? Organizations.index
GET /o/{name} Organizations.home
Authentication
GET /login Authentication.l
POST /login Authentication.
GET /register Authentication.
* /secure Authentication.
POST /logout Authentication.
Map static resources from the /app/public folder to the /publ
GET /public/ staticDir:publi
logout - Action
login - Action
Insert mode::



To install this plugin, copy the JAR file from **$PLAY_HOME/support/eclipse** to **$ECLIPSE_HOME/dropins**.

Play 1.0.1 is a maintenance release of the Play 1.0 branch. It fixes a lot of small bugs, provides a better binding for persistent JPA objects, auto-escaping of HTML and JavaScript code in templates to avoid XSS security problems.

> **Play 1.0.1** is a maintenance release and is fully compatible with the 1.0 version. If you encounter any problems please ask us on the Google Group.

You can read about the fixed bugs on the 1.0.1 road map page. The most important changes are highlighted in this page:

Automatic escaping for HTML code in templates

Now all dynamic expressions are escaped by the template engine to avoid XSS security issues in your application. So the **title** variable containing **<h1>Title</h1>** is now escaped:

```
${title} --> <h1>Title</h1>
```

If you really want to display it in an unescaped way, you need to explicitely call the **raw()** method:

```
${title.raw()} --> <h1>Title</h1>
```

Also, if you want to display a large part of raw HTML, you can use the **#{verbatim /}** tag:

```
#{verbatim}
    ${title} --> <h1>Title</h1>
#{/verbatim}
```

Of course, as this feature can break existing applications, it is not enabled by default. You can enable it by adding the following line to your **application.conf** file:

```
future.escapeInTemplates=true
```

This line is added to new applications created with the 1.0.1 release.

@javax.inject.Inject support

You can now use the **@Inject** annotation if you are using a Dependency Injection container like the one provided by the Spring module. The the **@Inject** annotation automatically injects defined beans in your controllers, jobs and mailer. It works on static fields.

For example to inject a **PriceWatcher** service defined in Spring to one of your controllers, just do:

```
public class Application extends Controller {

    @Inject
    static PriceWatcher prices;

    public static void index() {
        prices.getAll(); // prices is defined here
    }

}
```

Auto-reload should work as expected.

Better binding for JPA objects

Previously there was no way to automatically bind a JPA object using the HTTP to Java binding. For example:

```
public static void save(User user) {
    user.save(); // fail in 1.0
}
```

User being a JPA entity class, this code failed because user was a transient object created by the Binder and not yet managed by Hibernate.

The old way was to use the **edit()** method, like:

```
public static void save(Long id) {
    User user = User.findById(id);
    user.edit(params);
    user.save(); // ok
}
```

Now you can provide the **user.id** field yourself in the HTTP parameters. When Play finds the **id** field, it loads the matching instance from the database before editing it. The other parameters provided by the HTTP request are then applied. So you can save it directly.

```
public static void save(User user) {
    user.save(); // ok with 1.0.1
}
```

Of course as this feature can break existing applications it is not enabled by default. You can enable it by adding the following line to your **application.conf** file:

```
future.bindJPAObjects=true
```

This line is added to new applications created with the 1.0.1 release.

Framework ID can be set from the command line

The framework ID allows you to run different configurations for a same Play application (ie dev, test, staging, production, etc…).

Using the command line, you can now specify the application framework ID the application will use. For example, to run an application in production mode:

```
play run --%production
```

With these lines defined in the **application.conf** file:

```
application.mode=dev
%production.application.mode=prod
```

It is compatible with all the existing commands using the framework ID information. The default ID is still defined using the **play id** command.

By the way, **play test** is then equivalent to:

```
play run --%test
```

Custom validation

Can't find the validator you need in the **play.data.validation** package? Write your own. You can use the generic **@CheckWith** annotation to bind your own **Check** implementation.

For example:

```
public class User {

    @Required
    @CheckWith(MyPasswordCheck.class)
    public String password;

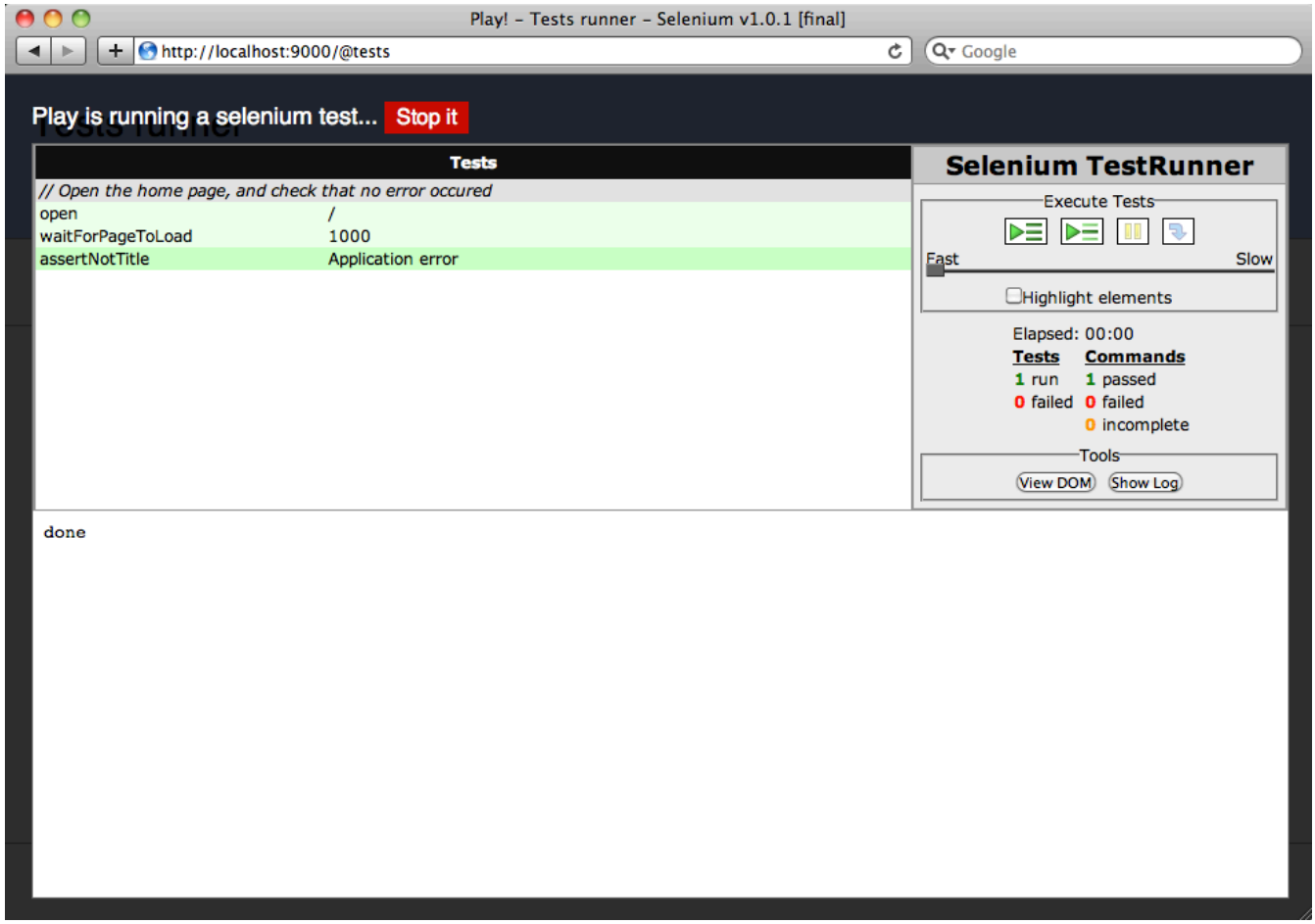
    static class MyPasswordCheck extends Check {

        public abstract boolean isSatisfied(Object user, Object password) {
            return notMatchPreviousPasswords(password);
        }

    }
}
```

Test runner update

We've updated Selenium to the 1.0.1 final version and improved the UI. Selenium tests now run in fullscreen. And some new functionality like the "Run all tests" have been added.



Also local documentation is now available when you run your application in test mode.

HTML5 as default doctype and JQuery in new applications

New applications come with a default HTML skeleton. Previously it was a **XHTML 1.0 Transitional** formatted HTML. Now it comes with a **HTML5** doctype by default:

```
<!DOCTYPE html>

<html>
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
        <link rel="stylesheet" type="text/css" media="screen"
href="@{'/public/stylesheets/main.css'}">
        #{get 'moreStyles' /}
        <link rel="shortcut icon" type="image/png"
href="@{'/public/file:///Users/nicolasleroux/Projects/Personal/1.1/documentation/images/
<script src="@{'/public/javascripts/jquery-1.3.2.min.js'}" type="text/javascript"
charset="utf-8"></script>
        #{get 'moreScripts' /}
    </head>
    <body>
        #{doLayout /}
    </body>
</html>
```

It's just the default and you can of course change it to any doctype if you wish. However HTML5 is surely the future and is way simpler.

The default template comes with the **JQuery** JavaScript library. This is because modern web applications need a good JavaScript framework and we love JQuery. Again it is just the default and you can of course override it with whatever JavaScript library you want.

Oh, and it comes with 2 default placeholders to inject even more scripts and/or styles. For example, add in your view:

```
#{set 'moreScripts'}
<script src="@{'/public/javascripts/gears_init.js'}" type="text/javascript"
></script>
#{/set}
```

And the gear JavaScript library will be accessible in your view.

Improved #{list} tag

The **items** parameter is now optional and can be replaced by the default **arg** argument.

So you can rewrite:

```
#{list items:users, as:'user'}
    <li>${user}</li>
#{/list}
```

as:

```
#{list users, as:'user'}
    <li>${user}</li>
#{/list}
```

The **as** parameter is now optional as well. It uses _ as default variable name:

```
#{list users}
    <li>${_}</li>
#{/list}
```

New #{jsAction /} tag

The **#{jsAction /}** tag allows to import a server-side defined route as a JavaScript function. It is very useful if you need to call an URL with an AJAX call using free variables.

Let's see an example:

```
GET     /users/{id}         Users.show
```

Now you can import this route on the client side:

```
<script type="text/javascript">
    var showUserAction = #{jsAction @Users.show(':id')}

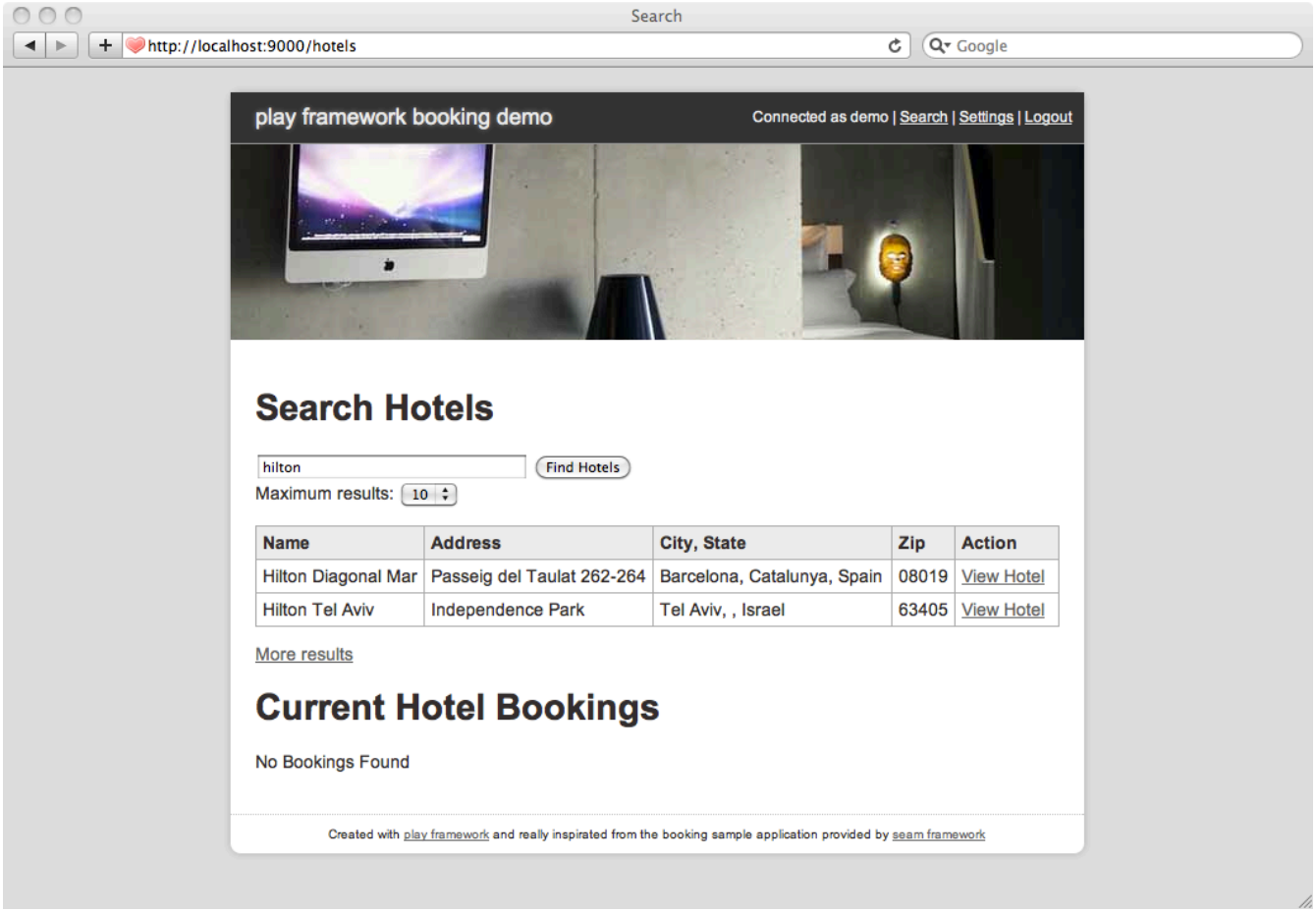
    var displayUserDetail = function(userId) {
        $('#userDetail').load( showUserAction({id: userId}) )
    }
</script>
```

As you see you use the **:name** syntax to declare free variables. You can mix free and fixed variables together. The **#{jsAction /}** tag generates a JavaScript function. This function takes a JavaScript object as parameter that must define all the free variables.

New sample application: 'booking'

The **booking** application is a port of the well known booking application from the JBoss Seam framework.

It demonstrates how you can manage complex states client side in a RESTful way with a stateless web framework.



Eclipse plug-in is on its way

We are working hard to provide a dedicated Eclipse plug-in. The current status is alpha but already usable. You can find it in the **support/eclipse** directory of your Play installation.

Application.java main.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
    <head>
        <title>#{get 'title' /}</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
        <link rel="stylesheet" type="text/css" media="screen" href="@{'/public/stylesheets/main.css'}" />
        #{get 'moreStyles' /}
        <script src="@{'/public/javascripts/jquery-1.3.2.min.js'}" type="text/javascript" charset="utf-8"></script>
        #{get 'moreScripts' /}
    </head>
    <body>

        <div class="result">
            <h1 class="${result.passed ? 'passed' : 'failed'}">${result.name.raw()}</h1>
        </div>

        <a class="revLink" href="${revisionDetail}">
            ${revisionDetail}
        </a>

        <ul>
            <li class="passed log">
                <a href="@{detail(result.id, 'application.log')}">application.log</a>
            </li>
            #{list result.tests(), as:'test'}
                <li class="${test.endsWith('.passed.html') ? 'passed' : 'failed'}">
                    <a href="@{detail(result.id, test)}">${test.replaceAll(/.(passed|failed).html/, '')}</a>
                </li>
            #{/list}
        </ul>

    </body>
</html>

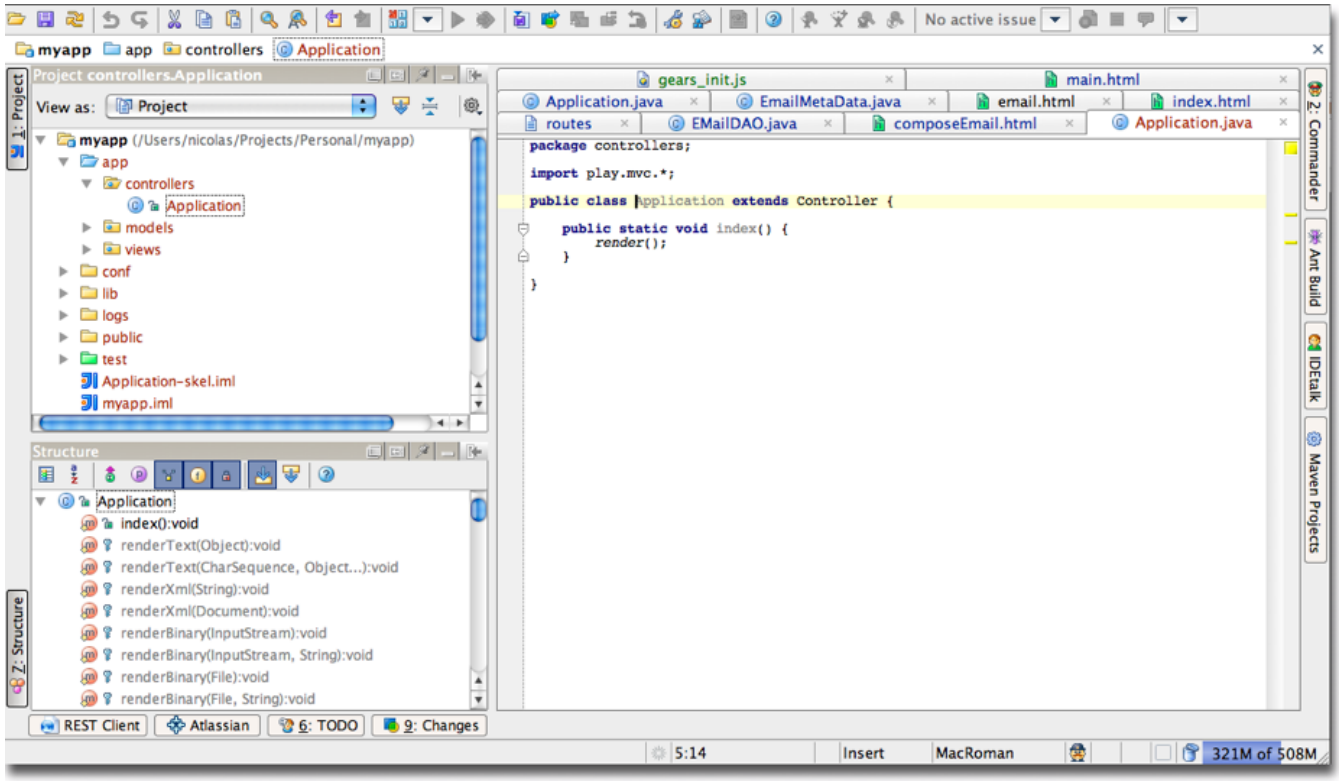
```

Support for IntelliJ IDEA

The Play framework now supports IntelliJ IDEA out of the box!

To convert your Play application into an IntelliJ IDEA project, use the **idealize** command:

```
# play idealize myApp
```



You can use the Run and Debug using the context menu.